Multilingual R5RS-Compliant Scheme Interpreter for JVMs

Viivi User's Manual

manual version 00.15.c[1]

(for Viivi version 00.15.02 and newer)

ilma

URL: <http://home.j00.itscom.net/ilma/>

e-mail: <ilma@j00.itscom.net>

May 3, 2012

[1]The manual is currently available in English and Japanese.

Copyright(c) 2010-2012 ilma <ilma@j00.itscom.net> All rights reserved.

Agreement

All rights relating to this manual belong to ilma <ilma@j00.itscom.net> the author of the software Viivi.

It is recommended to read this manual before using the software Viivi. This manual is in writing and editing and hence may contain errors. Before doing operation following to the descriptions in this manual, make enough confirmation that the operation is correct and suited to your environment. The author is exempt from the responsibility for any kind of disadvantage caused by this manual and related files or operations, whether the description in this manual is correct or not.

The user is expected to report errors and problems of this manual to the author when the user finds them. The copyright of the parts modified based on users' reports still belongs to the author.

This manual must be obtained directly from the download site of the author. This manual must be used in the form as it is distributed. Any kind of modification on any part of or whole of this manual is prohibited. Any kind of citation from any part of or whole of this manual without permission is prohibited. Any kind of redistribution of any part of or whole of this manual is prohibited.

This manual will be updated independently of the updates of the software Viivi. Check the download site as frequent as possible for the latest version of this manual. Agreement for the software Viivi is different from this agreement for the manual. Before using the software Viivi, please read the agreement attached to Viivi.

The product names of the hardwares and softwares appearing in this manual are registered trademarks of the respective makers.

When the contents of this agreement conflict between different description languages, Japanese description takes precedence over others.

Viivi release logs

User's Manual

1. viiviManual 00.00._a (2010/10/24) first released version
2. viiviManual 00.01._a (2010/11/23) periodic update
3. viiviManual 00.01._b (2010/12/11) minor revision
4. viiviManual 00.01._c (2010/12/16) minor revision
5. viiviManual 00.02._a (2010/12/30) periodic update
6. viiviManual 00.02._b (2011/01/18) minor revision
7. viiviManual 00.02._c (2011/01/20) minor revision
8. viiviManual 00.14.a (2012/02/21) periodic update
9. viiviManual 00.15.a (2012/03/18) periodic update
10. viiviManual 00.15.b (2012/03/23) minor revision (this version)

Package

1. viiviPackage.00.00.00.zip (2010/10/24) first released version
2. viiviPackage.00.00.01.zip (2010/10/25) minor revision
3. viiviPackage.00.00.02.zip (2010/10/26) minor revision
4. viiviPackage.00.00.03.zip (2010/10/29) minor revision
5. viiviPackage.00.00.04.zip (2010/10/30) minor revision
6. viiviPackage.00.00.05.zip (2010/10/31) minor revision
7. viiviPackage.00.00.06.zip (2010/11/02) minor revision
8. viiviPackage.00.00.07.zip (2010/11/10) minor revision
9. viiviPackage.00.00.08.zip (2010/11/10) minor revision
10. viiviPackage.00.00.09.zip (2010/11/13) minor revision
11. viiviPackage.00.01.00.zip (2010/11/23) periodic update
12. viiviPackage.00.01.01.zip (2010/11/23) minor revision
13. viiviPackage.00.01.02.zip (2010/11/24) minor revision
14. viiviPackage.00.01.03.zip (2010/11/30) minor revision
15. viiviPackage.00.01.04.zip (2010/12/08) minor revision
16. viiviPackage.00.01.05.zip (2010/12/16) minor revision
17. viiviPackage.00.02.00.zip (2010/12/30) periodic update
18. viiviPackage.00.02.01.zip (2011/01/18) minor revision
19. viiviPackage.00.03.00.zip (2011/01/28) periodic update
20. viiviPackage.00.04.00.zip (2011/02/28) periodic update
21. viiviPackage.00.05.00.zip (2011/03/30) periodic update
22. viiviPackage.00.05.01.zip (2011/04/16) minor revision
23. viiviPackage.00.06.00.zip (2011/04/29) periodic update
24. viiviPackage.00.07.00.zip (2011/06/01) periodic update
25. viiviPackage.00.07.01.zip (2011/06/07) minor revision
26. viiviPackage.00.08.00.zip (2011/07/01) periodic update
27. viiviPackage.00.09.00.zip (2011/08/10) periodic update
28. viiviPackage.00.09.01.zip (2011/08/11) minor revision
29. viiviPackage.00.10.00.zip (2011/08/31) periodic update
30. viiviPackage.00.10.01.zip (2011/09/02) minor revision
31. viiviPackage.00.10.02.zip (2011/09/08) minor revision
32. viiviPackage.00.11.00.zip (2011/10/02) periodic update
33. viiviPackage.00.12.00.zip (2011/11/05) periodic update
34. viiviPackage.00.13.00.zip (2011/12/21) periodic update
35. viiviPackage.00.13.01.zip (2011/12/29) minor revision
36. viiviPackage.00.13.02.zip (2011/12/30) minor revision
37. viiviPackage.00.14.00.zip (2012/01/31) periodic update

38. viiviPackage.00.14.01.zip (2012/02/21) minor revision
39. viiviPackage.00.14.02.zip (2012/03/05) minor revision
40. viiviPackage.00.15.00.zip (2012/03/18) periodic update

Notation in the manual

- Different character fonts are used for different descriptions:

Hello, everyone!	Normal character font is used for explanation.
`echo Hello true Hackers!`	URLs, File names, command strings, file contents, examples of screen display, etc. are shown with `Courier` font.
Viivi, ilma	Proper nouns concerning with the software that is explained in this manual are shown with Helvetica font.

- Simplifying notations are used in the explanations about Scheme procedure invocations and command line argument options of Viivi:

`(ABC)`	`ABC` can be omitted.
`[N]`	`N` can be omitted.
`FILE(S)`	Represents either `FILE` or `FILES`.
`[PQR|XYZ]`	Represents either `PQR` or `XYZ`.

To native speakers of English or German

The author's translation into English or German is poor and may be wrong. Would you be willing to correct wrong English or German in the messages output from Viivi and the descriptions on the manual/homepage? Thank you very much in advance.

Contents

1 Introduction to Viivi **6**
- 1.1 About this manual . . . 6
- 1.2 A tour-guide to this manual . . . 7
- 1.3 What is Viivi? . . . 8
- 1.4 Requirements for Viivi . . . 10
- 1.5 Confirmed platforms for Viivi . . . 11

2 Downloading and Installing Viivi **12**
- 2.0 Outline of Viivi installation processes . . . 12
- 2.1 Checking home directory . . . 12
- 2.2 Creating Viivi system directory . . . 12
- 2.3 Downloading Viivi distribution package . . . 13
- 2.4 Extracting Viivi distribution package file . . . 13
- 2.5 Trying to run Viivi: Stage-**A** . . . 14
- 2.6 Troubled? Try to assign memory capacity. . . . 15

3 Trying Viivi boot-up script file **17**
- 3.1 Selecting an appropriate Viivi boot-up script file . . . 17
- 3.2 Trying to run Viivi: Stage-**B** . . . 17
- 3.3 Troubled? Try to edit the boot-up script file . . . 17
  - 3.3.1 Editing points in the boot-up script file for Unix-like platforms . . . 18
  - 3.3.2 Editing points in the batch file for Windows platforms . . . 18
  - 3.3.3 Editing points common for all the platforms . . . 18

4 Separating Viivi work directory **20**
- 4.0 Outline of processes separating Viivi work directory . . . 20
- 4.1 Creating the user's work directory . . . 20
- 4.2 Setting the environmental variables . . . 21
- 4.3 Trying to run Viivi: Stage-**C** . . . 22
- 4.4 Troubled? Check and re-set the values of the environmental variables . . . 22

5 Editing Viivi Configuration File **23**
- 5.0 Outline of the processes editing the configuration file . . . 23
- 5.1 Copying the configuration file . . . 23
- 5.2 Editing the configuration file . . . 23
- 5.3 Trying to run Viivi: Stage-**D** . . . 23
- 5.4 Troubled? Do it over from copying the configuration file . . . 24

6 Viivi System Files **25**
- 6.1 `00readme*.txt` . . . 25
- 6.2 `Viivi.jar` . . . 25
- 6.3 `viivi` and `viivi.bat` . . . 25
- 6.4 `viivi.conf` and `.viivirc` . . . 27
  - 6.4.1 Hierarchy of the configuration files . . . 27
  - 6.4.2 Contents of the configuration files . . . 29

7 Viivi predefined constants **30**
7.1 Numerical constants . . . 30
7.2 Symbol constants . . . 30

8 Viivi predefined procedures **31**
8.1 Procedures to show information . . . 32
8.2 Procedures to obtain or to set properties of Viivi . . . 33
8.3 Procedures for random numbers . . . 45

9 Viivi command line argument options **46**
9.1 Command line argument options to show information . . . 46
9.2 Command line argument options for R5RS incompatibilities . . . 47
9.3 Command line argument options for specifying Viivi's behaviors . . . 49
9.4 Command line argument options for input-source/output-destination . . . 54

10 R5RS specifications of Viivi **59**
10.1 Equality testing procedures of Viivi . . . 59
10.2 General timing of argument evaluation in a procedure invocation . . . 60

11 Unique extensions of Viivi **61**
11.1 Viivi debug trace mechanism . . . 61
11.2 Configuring evaluation environment with base level . . . 70
11.3 Operations on circular lists . . . 73
11.4 Display styles for objects . . . 74
11.5 Direct input for objects . . . 75
11.6 Investigating macro pattern matching with the longest matching method . . . 77
11.7 To use a transcript output as another input . . . 80

12 Other tips for using Viivi **81**
12.1 To shut down Viivi . . . 81
12.2 To stop and restart screen scrolling . . . 81
12.3 Standard input/output files . . . 81

13 Reporting bugs of Viivi **82**
13.1 How to report . . . 82
13.2 Things to be reported (examples) . . . 82

14 Updating Viivi **83**
14.1 Update of the system files . . . 83
14.2 Update of the copyright string . . . 83

15 De-installing Viivi **85**
15.1 Deleting Viivi system files . . . 85
15.2 Deleting Viivi system directory . . . 85
15.3 Delete user's work directory . . . 86
15.4 Retracting Viivi environment variable settings . . . 86

1 Introduction to Viivi

1.1 About this manual

This manual summarizes information necessary for using Viivi.

Please visit Viivi's page from ilma's homepage

<http://home.j00.itscom.net/ilma/>

for the latest information on Viivi.

Please read the file `00readmeEN.txt` or `00readmeJP.txt` included in the Viivi distribution package.

You need to accept the agreement of Viivi to use the software Viivi.

This manual does not explain about the Scheme programming language. For the Scheme programming language, see R5RS[1] the standard language definition, or textbooks. R5RS is downloadable, for example, from

<http://www.schemers.org/Documents/Standards/R5RS/>.

Please send your messages via e-mail to

<ilma@j00.itscom.net>.

Even when you don't have your own e-mail address, you can send e-mail to ilma through the Viivi's page.

[1] Revised[5] Report on the Algorithmic Language Scheme

1.2 A tour-guide to this manual

- To grasp the outline of Viivi ...

  ⇒ Please continue reading this Section 1.

- To install Viivi ...

  ⇒ The procedures described Sections from 2 to 5 fully[2] install Viivi.

  However, it is still possible for you to continue to use Viivi even if you suspend the installation sequence at the end of an arbitrary section, when you don't understand what to do with the remaining procedures, or when you don't want to proceed any more from some reason[3].

- To know about the files in the Viivi package ...

  ⇒ Please read Section 6.

- To know the functions and the usage of Viivi ...

  ⇒ Please read Sections from 7 to 12.

  Note that, without reading the detailed descriptions, you will be able to use Viivi as an R5RS-compliant Scheme interpreter in an interactive way.

- To make bug-reports or comments on Viivi ...

  ⇒ Please read Section 13.

- To update or delete Viivi ...

  ⇒ Please read Section 14 or Section 15.

[2]The word "fully" means that the installation procedures allow you to boot up Viivi (1) only with the command `viivi` (2) in an arbitrary directory (3) without showing the agreement at the boot-up time.

[3]In that case, some inconvenience will be left about (1), (2), or (3).

1.3 What is Viivi?

Viivi[4] is a multi(-natural)-lingual R5RS compliant Scheme programming language interpreter for JVMs[5]. It has been implemented as ilma's private project using Java programming language from scratch based on a unique Object design aiming to be fully compliant with R5RS.

Viivi has the following features:

- Specs on the core of the R5RS language processor
  - Uniquely designed and implemented with no references at all to the internals of other implementations.
  - Being written with Pure Java language, it is expected to run similarly on any kinds of platform equipped with a JVM, only if memory storage is large enough. Supplementary Scheme programs are not required.
  - Fully supports the features defined in R5RS, including multiple generations and restarts of continuations.
  - All the numerical types defined in R5RS (integers of unlimited length, rational numbers, floating point numbers [64bit-precision], and complex numbers) are supported.
  - Numerical calculations are performed as exactly as possible (e.g., when both the real and the imaginary parts of the given complex numbers are all rational, then arithmetic calculations are performed keeping both the parts rational; when all the complex numbers are given in the polar format, then multiplications and divisions are performed in the form of polar format without being expanded into Descartes format).
  - `scheme-report-environment` and `null-environment` accept only an argument of `5`. Namely, only R5RS is currently supported. Older and newer versions, such as R4RS or R6RS, are not supported.
  - Arguments (to be evaluated) for a procedure invocation are always evaluated from left to right.
  - Capital and small letters are distinguished in symbols.
  - Square brackets `[...]` and `#[...]` in addition to parentheses `(...)` and `#(...)` can be used to express pairs and vectors. They are unified to `(...)` and `#(...)` after they are read by the interpreter.
  - Display style of pairs and vectors follows srfi-38.
  - Since the implementation does not depend on the internal structure of the JVM but uses only the basic Java APIs and the exception handling, it must be easily transplanted into other Object-Oriented platforms.

[4] It was firstly presented as Gino (provisional) at Shibuya.lisp Technical Talk `#3` on July the 4th, 2009 `<http://shibuya.lisp-users.org/2009/07/04/sltt-3-tb/>`.

[5] Java Virtual Machines

- Specs on unique extensions
  - Supports multiple natural languages in a parallel way by operating character data through the JVM internal Unicode.
    Currently supports English, Japanese, and German.
    Characters of the current locale language can be used in symbols and strings. Messages for errors and warnings are shown automatically in the current locale language.
    (In case of Japanese locale, for example, HIRAGANA, KATAKANA, KANJI characters can be used and the messages are shown in Japanese.)
    The user can also specify to use a language different from the current locale language.
    Potentially supports all the languages that the JVM Unicode supports. Consumed memory storage does not increase even when the number of the supported languages increases.
  - Equipped with a unique debug trace system which visualizes evaluating processes (even continuations as well!).
  - Not only files but also network-ports can be used as I/O ports.
  - The evaluation environment can be flexibly customized at the boot time with the configuration files (`viivi.conf` and `.viivirc`) and command line argument options. Equivalent customization can be performed at runtime using a procedure `viivi-set!`.
  - Sequential input sources and output destinations can be specified on the command line.
  - Customizable for "R5RS-incompatible" Scheme codes widely existing over the world (such that invoke `eval` with only one argument, that invoke continuation without arguments, that expect newly bound values returned from `define`/`set!`, and etc.).
  - When multiple ellipses appear in series within a list or a vector in a macro pattern, the macro pattern matching is investigated based on the principle of the maximum length matching.
  - The transcript output is written in a format that will be used as input to the interpreter. It enables to redo the same evaluations later or on another network host.
  - Introduced a concept of "base-level" which enables to create evaluation environment of an arbitrary combination of predefined object groups (procedures and constants) classified in R5RS. On a base level, individual predefined objects can be added or deleted. This is a feature assumed for the purpose to generate a subset environment of R5RS and to conduct experiments in the environment.
  - Length of a circular list is defined as the negative of the number of the actual elements on the list. (e.g., the length of `#0=(10 20 30 . #0)` is `-3`.)
  - When an error occurs, detailed error information is displayed. When a problem is likely to occur (for example, a symbol bound to a predefined procedure is re-bound to something else, or, a circular list is given as an argument to a procedure expecting a proper list as the argument), then a warning is displayed.

1.4 Requirements for Viivi

Viivi requires the platform the following to run.

- JRE[6] of version 1.6.0 or newer

  This is a software environment including JVM for Java programs to run. It is prepared for various platforms (workstations, PCs, Mac, PDAs, and cellular phones, etc.). Nowadays many platforms are already equipped with JREs at the initial configurations. If it is not yet installed, please obtain the latest JRE and install it correctly. JREs for PCs are available from the web page of ORACLE <http://www.oracle.com/>. For the JREs for other platforms, please ask the makers of your hardwares and OSs.
  JDK[7] includes JRE.
  Please use JRE of 1.6.0 or newer version. Viivi does not run on JREs older than 1.6.0.
  Explanations in this manual assume that the JRE is already installed on the platform.

- enough memory storage

  The minimum memory storage that Viivi itself requires (to run the sample codes in R5RS, for example) is about

  4MB for the 32bit JVM and
  8MB for the 64bit JVM,

  respectively. Additional memory storage is practically required for OS and JVM.
  There is no upper limit for the memory storage that Viivi can use. The upper limit is restricted by the limitations of the hardware, OS, and JVM.
  (There is a result that Viivi used 48GB on a 64GB memory equipped machine.)

[6] Java Runtime Environment
[7] Java Development Kit

1.5 Confirmed platforms for Viivi

Here is a list of confirmed platforms for Viivi. Basically Viivi is expected to run anywhere independent of the platform, only if it satisfies "1.4 Requirements for Viivi".

If you find Viivi to run on other platforms, including PDAs and cellular phones, please let me know. They will be appended to this list.

Platforms in the following without special notations are confirmed by ilma.

- Windows XP + Sun Microsystems JRE 1.6.0_20-b02
  (Thanks to taayan, hiyoax, and ghost!)
- Windows Vista + Sun Microsystems JRE 1.6.0_20-b02 (32-bit/64-bit)
- Windows 7 + Sun Microsystems JRE 1.6.0_20-b02 (32-bit)
- Windows 7 + Sun Microsystems JRE 1.7.0_03-b05 (32-bit)
- Mac OS X 10.6.3 + Apple JRE 1.6.0_15-b03-219 (64-bit)
  (Thanks to pixie and sneex!)
- Fedora 12 + Sun Microsystems JRE 1.6.0_20-b02 (32-bit/64-bit)
- Ubuntu 9.10 Karmic Koala + Sun Microsystems JRE 1.6.0_20-b02 (32-bit/64-bit)
- Ubuntu 10.04 LTS Lucid Lynx + OpenJDK Runtime Environment 1.6.0_18-b18 (64-bit)
- Ubuntu 10.04 LTS Lucid Lynx + Sun Microsystems JRE 1.6.0_20-b02 (32-bit/64-bit)
- Ubuntu 11.04 Natty Narwhal + unknown JRE (64-bit)
  (Thanks to g000001!)
- Ubuntu 11.10 Oneiric Ocelot + Oracle JRE 1.6.0_30-b12 (32-bit/64-bit)
- Ubuntu 11.10 Oneiric Ocelot + Oracle JRE 1.7.0_03-b04 (32-bit/64-bit)
- OpenSolaris + Sun Microsystems JRE 1.6.0_20-b02 (32-bit/64-bit)
- OpenIndiana 151a + Oracle JRE 1.7.0_03-b04 (32-bit)
- Solaris 10 10/09 + Sun Microsystems JRE 1.6.0_20-b02 (32-bit/64-bit)
- Solaris 11 11/11 + Oracle JRE 1.6.0_31-b04 (64-bit)
- Solaris 11 11/11 + Oracle JRE 1.7.0_03-b04 (64-bit)

2 Downloading and Installing Viivi

In this section, downloading and installing processes of Viivi will be explained. The user is assumed to be familiar with the operations with files and directories[8] on the platform.

2.0 Outline of Viivi installation processes

Briefly speaking beforehand, the processes to install Viivi after downloading the package zip file are as follows:

i. Create a directory named `Viivi` in the home directory
("2.1 Checking home directory" and "2.2 Creating Viivi system directory")

ii. Extract the `Viivi` distribution package zip file into the created directory
("2.3 Downloading Viivi distribution package" and "2.4 Extracting Viivi distribution package")

Each process of downloading and installing Viivi is detailed in the following. Specific operations will be shown for some major platforms: Unix-like platforms with `sh` family (`sh`, `bash`, and `ksh`; including Mac OS), Unix-like platforms with `csh` family (`csh` and `tcsh`), and Windows platforms. Please do those operations, after confirming that they are correct and suited to your platform. For other kinds of platform, please do corresponding operations after confirming that they are correct and suited to your platform.

If your browser supports the feature of extracting a zip file at downloading time, you can make the processes `i.` and `ii.` simultaneously at the downloading time. If it is the case, proceed to "2.3 Downloading Viivi distribution package".

2.1 Checking home directory

First of all, please check your "home directory" on the platform. The home directory for a user whose account name is ilma, for example, may be as follows, depending on the platform types:

Examples	
`/home/ilma`	for Unix-like platforms
`/home/teamK/ilma`	for Unix-like platforms
	(They may be represented as `~` or `${HOME}` in the following)
`C:\Users\ilma`	for Windows platforms
`D:\ilma`	for Windows platforms

2.2 Creating Viivi system directory

In the home directory which you have checked in "2.1 Checking home directory", create the Viivi system directory named `Viivi`:

[8] The word "folder" is usually used for the Windows platforms. In this manual, however, the word "directory" is generally used both for Unix-like and Windows platforms. The word "folder" is also used in explanations only for the Windows platforms.

To create the Viivi system directory using the Unix shells or Windows `cmd`, do the following in your home directory:

`$ mkdir Viivi`	for Unix-like platforms with `sh` family (`$` is a shell prompt)
`% mkdir Viivi`	for Unix-like platforms with `sh` family (`%` is a shell prompt)
`> md Viivi`	for Windows platforms (`>` is a `cmd` prompt)

If you are familiar with GUI (Graphical User Interface) operation, you may create the Viivi system directory in your home directory using GUI on your platform.

2.3 Downloading Viivi distribution package

Download the latest version of the Viivi distribution package from the download site with the top-page

`<http://home.j00.itscom.net/ilma/>`.

The filename of the Viivi distribution package is as

`viiviPackage.XX.YY.ZZ.zip`.

The part `XX.YY.ZZ` represents the version number. Always use the latest version.

If your browser supports the function of extracting a zip file at downloading time, you can do the installation processes of "2.2 Creating Viivi system directory" and "2.4 Extracting Viivi distribution package file" simultaneously. At the downloading time, create a directory named `Viivi` in the home directory and extract the package zip file into it. And then proceed to "2.5 Trying to run Viivi: Stage-**A**".

If you don't extract the Viivi distribution package zip file at the downloading time, save the zip file directly into the Viivi system directory created in "2.2 Creating Viivi system directory".

If you already saved the Viivi distribution package zip file into another directory, move it to the Viivi system directory.

2.4 Extracting Viivi distribution package file

In the Viivi system directory, use `unzip` command to extract the Viivi distribution package zip file as

`$ unzip viiviPackage.XX.YY.ZZ.zip`	for Unix-like platforms with `sh` family
`% unzip viiviPackage.XX.YY.ZZ.zip`	for Unix-like platforms with `csh` family
`> unzip viiviPackage.XX.YY.ZZ.zip`	for Windows platforms.

If you are familiar with GUI (Graphical User Interface) operation, you may extract the Viivi distribution package file using GUI.

If you have succeeded in the file extraction, you will find in the Viivi system directory nine Viivi system files

`00readmeEN.txt`	attached read-me text file (English)
`00readmeJP.txt`	attached read-me text file (Japanese)
`Viivi.jar`	main body of the software Viivi
`viivi`	boot-up script file for Unix-like platforms
`viivi.ORG`	back-up file for `viivi`
`viivi.bat`	boot-up batch file for Windows platforms
`viivi.bat.ORG`	back-up file for `viivi.bat`
`viivi.conf`	configuration file
`viivi.conf.ORG`	back-up file for `viivi.conf`.

Don't forget to read `00readme[EN|JP].txt` file. It may contain important information which is not yet written on the User's manual.

2.5 Trying to run Viivi: Stage-A

Now Viivi must run. Let's try to boot it up.
In the Viivi system directory, input as follows:

`$ java -jar Viivi.jar`	for Unix-like platforms with `sh` family
`% java -jar Viivi.jar`	for Unix-like platforms with `csh` family
`> java -jar Viivi.jar`	for Windows platforms

If Viivi booted up correctly, you see agreement on the display. Please read it[9].

If you see Viivi's prompt

```
>
```

at the lowest line on the screen, Viivi is now running.

When you see some error message despite that JRE is correctly installed, try the operation in "2.6 Troubled? Try to assign memory capacity".

Now, let's try to input an expression `30`. Type `3` and next `0` from keyboard or other input device, and then push `Enter` or `Return` key or do equivalent action.

If the screen displays like

```
> 30
30
>
```

then Viivi is working correctly.
This shows that an S-expression `30` was read by the interpreter and that a value `30` was returned.

After Viivi is confirmed to run, try to escape from the interpreter by inputting `(exit)` (Don't forget to push `Enter`/`Return` key following to it).

[9]Seeing the agreement every time must be annoying. We will configure it afterwards not to show it at the boot time. For a while, you can suppress the agreement by giving Viivi a command line argument option `-v 4` at the boot time:

```
$ java -jar Viivi.jar -v 4
```

```
> (exit)
... A farewell message is shown here.
$                                          for Unix-like platforms with sh family
```

The prompt of the platform appearing below the farewell message shows that you have escaped from Viivi.

When you cannot escape from Viivi, you can shut Viivi down by killing JVM with `[Control]+[C]`[10] on most platforms.

Once Viivi works correctly, you may delete the Viivi distribution package zip file (`viiviPackage.XX.YY.ZZ.zip`).

For the time being, you are able to boot up Viivi any time you like: Go into the Viivi system directory and input the command you confirmed here (such as `java -jar Viivi.jar`).

In the next section "3 Viivi boot-up script", the boot-up script is used to make it easier to boot up Viivi.

2.6 Troubled? Try to assign memory capacity.

In case that an error message such as

```
Error occurred during initialization of VM
```
... (See below for the second-line error message)...

appears when you try to boot up Viivi, you need to assign the memory capacity for Viivi.

The measures depend on the second-line error message:

- "`Too small initial heap for new size specified`"
  You may need to increase the memory capacity for Viivi.
- "`Could not reserve enough space for object heap`"
  You may need to decrease the memory capacity for Viivi.

Give command line arguments to the `java` command to assign the memory capacity for Viivi.

```
$ java -Xss64m -Xms64m -Xmx256m -jar Viivi.jar      for Unix-like platforms (sh family)
% java -Xss64m -Xms64m -Xmx256m -jar Viivi.jar      for Unix-like platforms (csh family)
> java -Xss64m -Xms64m -Xmx256m -jar Viivi.jar      Windows platforms
```

The first three arguments given to the `java` command represent memory capacity assignments for different purposes:

`-Xss64m`	to assign 64MB for the JVM stack
`-Xms64m`	to assign 64MB for the JVM minimum heap
`-Xmx256m`	to assign 256MB for the JVM maximum heap

[10] pushing `C`-key with `Control`-key of the keyboard

You may need to decrease each memory capacity smaller for the platforms with smaller memory capacities (such as PDAs and cellular phones).

The author tried to reduce the memory capacity assignments to the minimum values on the PC platform (OSs are Solaris and Ubuntu). Depending on whether the JVM is 32bit or 64bit, the minimum values were about

```
java -Xss2m -Xms2m -Xmx2m -jar Viivi.jar        for 32-bit JVM
java -Xss4m -Xms4m -Xmx4m -jar Viivi.jar        for 64-bit JVM,
```

respectively. Please try to adjust the values for your platform so as to boot up Viivi on it.

3 Trying Viivi boot-up script file

In this section, the boot-up script file in the Viivi system directory is tried to boot up Viivi. With the boot-up script file, you can boot-up Viivi with fewer strokes of typing. Furthermore, command line arguments for `java` command can be written in the boot-up scripts file to specify the bit model of the JVM or to assign memory capacities for the JVM.

3.1 Selecting an appropriate Viivi boot-up script file

There are two kinds of script file to boot up Viivi in the Viivi system directory for the major platforms:

`viivi`	a `sh` script file for Unix-like platforms
`viivi.bat`	a batch file for Windows platforms

We call them as "boot-up script files" in the following. Select one appropriate to your platform. Even in case that you have a platform which does not run these script files, it must not be difficult to write your own boot-up script file based on the existing ones.

3.2 Trying to run Viivi: Stage-B

The distributed boot-up script files must work as they are for most platforms. After moving into the Viivi system directory, try to run the boot-up script file by inputting the following command to the prompt of the platform

`$ ./viivi`	for Unix-like platforms (`sh` family)
`% ./viivi`	for Unix-like platforms (`csh` family)
`> .\viivi`	for Windows platforms.

If Viivi's prompt

```
>
```

appears after the agreement as you saw in "2.5 Trying to run Viivi: Stage-**A**", then you can continue to use the boot-up script file. After checking that correct values are returned by inputting `30` or something else, quit by inputting `(exit)`.

In case that some error occurs in spite of the correct operations with the appropriate boot-up script file, you may need to make some modifications on it. Please proceed to the next section "3.3 Troubled? Try to edit the boot-up script file".

When the boot-up script file works for booting-up Viivi, please proceed to "4 Separating Viivi work directory" to use Viivi in a working directory different from the Viivi system directory.

3.3 Troubled? Try to edit the boot-up script file

Try to accommodate the boot-up script file to boot-up Viivi correctly, by editing the following applicable points.

In case of spoiling the boot-up script file by editing it in a wrong way, you can do it over from the

beginning by removing the kaput file and then copying the backup file (that with the filename ending with `.ORG`) to a new unedited boot-up script file.

3.3.1 Editing points in the boot-up script file for Unix-like platforms

- Does the boot-up script file have an executable attribute? If not, set it with the command:

```
$ chmod ugo+x viivi
```

- Is the path following to `#!` on the first line in the boot-up script file correctly set to the shell? If not, edit it correctly as

Examples
`#!/bin/bash`
`#!/usr/local/bin/tcsh`
etc.

3.3.2 Editing points in the batch file for Windows platforms

The batch file must work as it is for the Windows platforms. Check the batch file `viivi.bat` if it is not kaput.

3.3.3 Editing points common for all the platforms

If you see the error messages such as

```
Error occurred during initialization of VM
```
... (See below for the second-line error message)...

just after you run the boot-up script, you may need to change the assignments of the memory capacity for Viivi.

The measures depend on the second-line error message:

- "`Too small initial heap for new size specified`"
  You may need to increase the memory capacity assigned for Viivi.

- "`Could not reserve enough space for object heap`"
  You may need to decrease the memory capacity assigned for Viivi.

Each script file has a main line assigning the memory capacities. The boot script file `viivi` for Unix-like platforms is exampled in the following.

```
java -Xss64m -Xms64m -Xmx256m -jar ${VIIVI_HOME}Viivi.jar ${*}
```

The first three arguments for the command `java` represent memory capacity assignments for different purposes:

`-Xss64m`	to assign 64MB for the JVM stack
`-Xms64m`	to assign 64MB for the JVM minimum heap
`-Xmx256m`	to assign 256MB for the JVM maximum heap

Please try to adjust the values for your platform so as to boot up Viivi correctly.

You may need to decrease each memory capacity smaller for the platforms with smaller memory capacities (such as PDAs and cellular phones).

The author tried to reduce the memory capacity assignments to the minimum values on the PC platform (using Solaris and Ubuntu).
The minimum values to boot up Viivi were about:

```
java -Xss2m -Xms2m -Xmx2m -jar Viivi.jar          for 32-bit JVM
java -Xss4m -Xms4m -Xmx4m -jar Viivi.jar          for 64-bit JVM
```

The main line assigning the minimum memory capacity for 32-bit JVM is written in the boot-up script file in a commented form:

```
# java -Xss2m -Xms2m -Xmx2m -jar ${VIIVI_HOME}Viivi.jar ${*}
```

Uncomment it by removing the `#` at the head of the commented line, and comment the currently active line by putting a `#` at the head, and you can use the boot-up script file for minimum memory configurations. (This is an example for Unix-like platforms. Use `rem` in place of `#` to make a line commented for Windows platforms.)

When you assigned small memory capacities, of course, Scheme programs which requires large memory capacities does not run. Tune the values of memory assignment for your platform and your purpose through the practical experience with Viivi.

Once you succeeded in booting-up Viivi with the customized boot-up script file, it is recommended to make a backup file for it, so as not to make it overwritten at the update time. To do so, enter the following command in the Viivi system directory:

```
$ cp viivi viivi.SAVE                  for Unix-like platforms with sh-family
% cp viivi viivi.SAVE                  for Unix-like platforms with csh-family
> copy viivi.bat viivi.bat.SAVE        for Windows platforms
```

If you are familiar with GUI operation, you may create the backup file using GUI.

See "6.3 `viivi` and `viivi.bat`" for details of the boot-up script files.

4 Separating Viivi work directory

In this section, a work directory (where the user tries Viivi) will be separated from the Viivi system directory (where the system files Viivi resides) so as to use Viivi safely and conveniently.

In this manual, these two directories are assumed as:

- `Viivi` system directory
  a directory named `Viivi` in the home directory (You have already created it.)
- the user's work directory for Viivi
  a directory named `ViiviWork` in the home directory (You are going to create it.)

4.0 Outline of processes separating Viivi work directory

Briefly speaking the processes beforehand,

i. Create a directory named `ViiviWork` in the home directory.
   ("4.1 Creating Viivi work directory")

ii. Set the environmental variable `VIIVI_HOME` to the path string for the Viivi system directory (Don't forget adding a path-separator at the end of the path string!).
   ("4.2 Setting the environmental variables")

iii. Append the path string for the Viivi system directory to the end of the value of the environmental variable `PATH`. ("4.2 Setting the environmental variables")

Each process will be detailed in the following.

4.1 Creating the user's work directory

In the home directory which you checked in "2.1 Checking home directory", create the user's work directory named `ViiviWork` for trying Viivi:

To create the user's work directory on the Unix shells or Windows `cmd`, do the following in your home directory:

`$ mkdir ViiviWork`	for Unix-like platforms (`sh` family)
`% mkdir ViiviWork`	for Unix-like platforms (`csh` family)
`> md ViiviWork`	for Windows platforms

If you are familiar with GUI (Graphical User Interface) operation, you may create the user's work directory for Viivi using GUI on your platform.

4.2 Setting the environmental variables

In order to boot up Viivi from the user's work directory, the environmental variables of the platform `VIIVI_HOME` and `PATH` will be set to point the Viivi system directory.

Note that a path-separator (`/` and `\` for Unix-like platforms and Windows platforms, respectively) is needed at the end of the path string set to the environmental variable `VIIVI_HOME`.

If you are familiar with GUI operation, you may set the environmental variables using GUI on your platform.

- Unix-like platforms (`sh` family)

  Append the following four lines at the end of the environment configuration file (`.profile`, `.bashrc`, or `.kshrc`) in the home directory[11]:

  ```
  VIIVI_HOME=~/Viivi/                  #Don't forget / at the end of the line.
  export VIIVI_HOME
  PATH=${PATH}:${VIIVI_HOME}
  export PATH
  ```

  And then read in the environment configuration file to activate the edited environmental settings:
  `$ . ~/.profile` or `$ . ~/.bashrc` or `$ . ~/.kshrc`

- Unix-like platforms (`csh` family)

  Append the following two lines at the end of the environment configuration file (`.login` or `.cshrc`) in the home directory:

  ```
  setenv VIIVI_HOME ~/Viivi/                  #Don't forget / at the end of the line.
  setenv PATH ${PATH}:${VIIVI_HOME}
  ```

  And then read in the environment configuration file to activate the edited environmental settings:
  `$ source ~/.login` or `$ source ~/.cshrc`

- Windows platforms

  For Windows platforms, the standard GUI method is shown here. If you know how to set the environmental variables in the Windows platforms by editing system files directly, you may do it at your own risk.
  (The following assumes the user's home folder as `C:\Users\ilma`.
  Replace the underlined parts according to your environment.)

  ```
  [Start]-->[Control Panel]-->([System and Maintenance])-->[System]-->
  [Tasks]-->[Advanced System Settings]-->[Environment Variables]-->[New]
  ```

[11] For Mac OS, using `bash` by default, `.bashrc` seems to be appropriate.

Variable	Value
VIIVI_HOME	C:\Users\ilma\Viivi\
PATH	existing-path-strings-here;%VIIVI_HOME%

Don't forget "\" at the end of the line.

Then select -->[OK] and reboot Windows to activate the new settings.

4.3 Trying to run Viivi: Stage-C

Now you must be able to boot up Viivi from any directory. Move to the user's work directory and try to boot up Viivi. If you see agreement after doing the following operations, you have succeeded in separating Viivi work directory.

- Unix-like platforms (sh family)
  The boot-up script file is viivi.

```
$ cd ~/ViiviWork
$ viivi
```

- Unix-like platforms (csh family)
  The boot-up script file is viivi.

```
% cd ~/ViiviWork
% viivi
```

- Windows platforms
  The boot-up batch file is viivi.bat.
  (The following assumes the user's home folder as C:\Users\ilma.
  Replace the underlined part according to your environment.)

```
> C:
> cd \Users\ilma\ViiviWork
> viivi
```

4.4 Troubled? Check and re-set the values of the environmental variables

If Viivi does not boot up, check the values of the environmental variables VIIVI_HOME and PATH if they are correctly set. If they are not correctly set, re-set them with the processes described in "4.2 Setting the environmental variables".

5 Editing Viivi Configuration File

This section explains how to edit the Viivi configuration file for customizing Viivi. The agreement appearing at every boot time must be annoying the most users. Let's customize Viivi to suppress the opening message.

5.0 Outline of the processes editing the configuration file

Briefly speaking the processes beforehand,

i. Copy the configuration file `viivi.conf` in Viivi system directory into the user's work directory. ("5.1 Copying the configuration file")

ii. Edit the configuration file `viivi.conf` in the user's work directory. ("5.2 Editing the configuration file")

Each process will be detailed in the following.

5.1 Copying the configuration file

The original configuration file for Viivi is the file `viivi.conf`. Don't edit this original file but leave it as it is. The editable configuration file is made by copying the original file into the user's work directory.

To make a copy of the original configuration file `viivi.conf` into the user's work directory, do the following operation in the user's work directory:

```
$ cp ../Viivi/viivi.conf .          for Unix-like platforms (sh family)
% cp ../Viivi/viivi.conf .          for Unix-like platforms (csh family)
> copy ..\Viivi\viivi.conf .        for Windows platforms
```

If you are familiar with GUI (Graphical User Interface) operation, you may copy the configuration file `viivi.conf` using GUI on your platform.

5.2 Editing the configuration file

Edit the configuration file `viivi.conf` in the user's work directory with your favorite editor at the following part:

```
(viivi-set!  verbose-level 5) ; replace 5 with 3 to suppress opening message
```

in the original form into

```
(viivi-set!  verbose-level 3) ; replace 5 with 3 to suppress opening message
```

and then save (overwrite) it with the same name.

5.3 Trying to run Viivi: Stage-D

Try to boot up Viivi, and if you see only Viivi's prompt

```
>
```

without preceding messages, then you have succeeded in customizing Viivi. From now on, you can

boot up Viivi without displaying the opening messages.

Congratulations!
Now you have completed all the processes of Viivi installation. You will be able to boot up Viivi any time you like using the command `viivi`. Remember that you are strongly recommended to try Viivi in the user's work directory to reduce risks.

See "6.4 `viivi.conf` and `.viivirc`" for details of the configuration files.

5.4 Troubled? Do it over from copying the configuration file

If you failed in editing the configuration file `viivi.conf` in the user's work directory and Viivi does not boot, then you can start it over from "5.1 Copying the configuration file".

6 Viivi System Files

"Viivi system files" are the files included in the Viivi distribution package. Each of them is explained in this section.

6.1 00readme*.txt

Text files including a brief explanations and the latest information about Viivi and the distribution package.

`00readmeEN.txt`	read-me file written in English
`00readmeJP.txt`	read-me file written in Japanese

These files are not required to run Viivi. But remember to read it, as they may contain latest information that is not yet described in the user's manual. The character code is UTF-8. If you cannot read the file with the characters garbled, you can read it with Web browsers or with LibreOffice[12]/OpenOffice[13] by setting the character encoding to UTF-8.

6.2 Viivi.jar

This is the main body of the Viivi software. User is not permitted to extract the `Viivi.jar` file or to operate the contents of the file. Use it as it is.

6.3 viivi and viivi.bat

They are a script file and a batch file for Unix-like platforms and Windows platforms, respectively, to boot up Viivi. The user may edit these files for the user's environments. Leave the backup files (of the filenames ending with `.ORG`) as they are.

`viivi`	a shell script file for Unix-like platforms to boot up Viivi
`viivi.ORG`	a back-up file for `viivi`
`viivi.bat`	a batch file for Windows platforms to boot up Viivi
`viivi.bat.ORG`	a back-up file for `viivi.bat`

Once the boot-up script is edited to run correctly, it is strongly recommended to make a copy of it with the filename extension `.SAVE` (with a name such as `viivi.SAVE` or `viivi.bat.SAVE`) to avoid being overwritten in the processes updating to newer versions.

The purpose of these files is to make it easier to boot-up Viivi. At the same time, command line argument options for the JVM, specifying bit model of the JVM and memory storage assignments for the JVM, can be written in the files as well.

[12] `http://www.libreoffice.org/`
[13] `http://www.openoffice.org/`

In both files, the most important part is the line beginning with `java`:

`viivi`:
```
java -Xss64m -Xms64m -Xmx256m -jar ${VIIVI_HOME}Viivi.jar ${*}
```

`viivi.bat`:
```
java -Xss64m -Xms64m -Xmx256m -jar %VIIVI_HOME%Viivi.jar %1 %2 ...  %8 %9
```

- The first word `java` is a command to start the JVM.
- The first three arguments for `java` specify the memory capacity assignments for the JVM. The underlined parameters below show values of the memory storages.
  If Viivi does not run correctly, tune these values suited to your environment.
  - The first argument `-Xss64m` is for the stack size of the JVM.
    Here 64MB is assigned to the stack.
    Increasing this value makes it effective for programs of deep procedure calls.
  - The second argument `-Xms64m` is for the initial heap size of the JVM.
    Here 64MB is assigned to the initial heap.
    Decreasing this value makes it cost-effective for programs of small data amount.
  - The third argument `-Xmx256m` is for the maximum heap size of the JVM.
    Here 256MB is assigned to the maximum heap.
    Increasing this value makes it effective for programs of large data amount.
- The fourth and the fifth arguments, `-jar ${VIIVI_HOME}Viivi.jar` and `-jar %VIIVI_HOME%Viivi.jar` in each file, respectively, indicate that the JVM runs `Viivi.jar` software in the Viivi system directory.
- The following arguments, `${*}` and `%1 %2 %3 %4 %5 %6 %7 %8 %9` in each file, respectively, are replaced with the arguments that user gave to the boot-up script.
- If 64-bit mode JVM is available on your platform, you can insert `-d64` as the first argument for `java` possibly to improve performance and to increase available memory storage.
- In the boot-up script file `viivi` for Unix-like platforms, the path to the shell that execute the boot-up script is written at the top line. It is

  ```
  #!/bin/bash
  ```

  by default[14]. If it does not work, rewrite the path pointing to the right shell at the top line (into such as `/bin/sh` or `/usr/local/bin/tcsh`, etc.).

You could write in the boot-up script file (`viivi` or `viivi.bat`) arguments for customizing Viivi. However, it is strongly recommended to write them in the configuration files (`viivi.conf` and `.viivirc`) as far as equivalent option settings are available.

[14]The default shell was changed from `/bin/sh` to `/bin/bash`, as `bash` seems to be available in most of the modern platforms.

Please obey the following policy for the locations specifying options to customize Viivi.

1. Options for the JVM should be specified in the boot-up script files `viivi` or `viivi.bat`.
2. Permanent or long-term options for Viivi should be specified in the configuration files `viivi.conf` or `.viivirc`.
3. Frequently used, temporary or short-term options for Viivi should be specified in normal Scheme files and should be read with the command line argument option `-L`.
4. Rarely used, temporary or short-term options for Viivi should be specified directly as command line argument options.

For example, it is NOT recommended to specify the verbosity option for Viivi in the boot-up script file `viivi` as

```
java -Xss64m -Xms64m -Xmx256m -jar ${VIIVI_HOME}Viivi.jar -v 3 ${*}
```

Instead, specify it in the Viivi configuration file `viivi.conf` as[15]

```
(viivi-set!  verbose-level 3)
```

6.4 `viivi.conf` and `.viivirc`

Configuration files `viivi.conf` and `.viivirc` are for users to customize Viivi's behavior at the boot time.

The Viivi distribution package includes `viivi.conf` file but no `.viivirc` file.
The contents of `viivi.conf` and `.viivirc` are the same. And hence the user can make the file `.viivirc` as a copy of the file `viivi.conf`. The only one difference is that the former is visible but the latter is invisible in the default environments of Unix-like platforms.

The contents of the configuration files are Scheme language programs. But available procedures are limited and the evaluation processes and the results will never be shown.

The file `viivi.conf.ORG` is a backup file for the configuration file `viivi.conf`. Leave the backup file as it is.

6.4.1 Hierarchy of the configuration files

Multiple configuration files can exist simultaneously. Only the configuration file `viivi.conf` in the Viivi system directory is indispensable. Other configuration files are optional. They are read in the order from upper to lower in the list shown below. Lower configuration files are prior to the upper files in affecting the state of Viivi at the boot time.

[15] If it were not for such a policy, specifications on Viivi's behavior would be dispersed over files and the user would be confused. In the above example, when the verbosity specifications were written in the both files, that written in the boot-up script file is prior to that in the configuration file (because Viivi at last receives the former as a command line argument option set). But it is not easy for the user to grasp such priority. To avoid this kind of confusion, specifications written in the boot-up script files should be restricted to those for the JVM.

In the following, ${VIIVI_HOME}, ${HOME}, and ${PWD} represent the Viivi system directory (appended a path separator at the end), the user's home directory, and the directory where user boots up Viivi. Provided that user always tries Viivi in the user's work directory, ${PWD} is the user's work directory.

0. ${VIIVI_HOME}viivi.conf (indispensable, uneditable)
1. ${HOME}/.viivirc (optional, editable)
2. ${HOME}/viivi.conf (optional, editable)
3. ${PWD}/.viivirc (optional, editable)
4. ${PWD}/viivi.conf (optional, editable)

Don't modify but leave the original configuration file ${VIIVI_HOME}viivi.conf as it is.

Strictly speaking, the files from 0. to 4. are read sequentially as configuration files just before Viivi is booted up.

You can provide an arbitrary Scheme file with the command line argument option -L

5. a name of a usual Scheme file following to -L on the command line (optional, editable)

Strictly speaking, the file provided as 5. is read just after Viivi is booted up as the first library file. You can provide multiple library files in sequence as well, with -L option before each filename, on the command line.

And finally, specifications given as the command line argument options are read, although they are not configuration files.

[6. specifications given as the command line argument options (optional)]

User can flexibly customize Viivi by specifying default settings in the files of lower priority (upper in the list) and settings specific to the user or the directory for the project in the files (and command line argument options) of higher priority (lower in the list).

You may have noticed that, besides the uneditable file ${VIIVI_HOME}viivi.conf, there are two editable configuration files viivi.conf and .viivirc in the home directory ${HOME} and the current working directory ${PWD}, respectively. As viivi.conf has higher priority than .viivirc, these two configuration files can be distinguished as follows:

- New configuration items that are being tested should be written in the configuration file viivi.conf first.
- After the configuration items are confirmed to work correctly, the configuration items should be moved into the other configuration file .viivirc.

6.4.2 Contents of the configuration files

Notation in the configuration files follows that of the Scheme programming language, including the comment notation.

Only the following three procedures can be used in the configuration files:[16]

`viivi-set!`	a procedure to modify properties of Viivi interpreter ("8 Viivi predefined procedures").
`random-seed`	a procedure to set a seed for pseudo-random number generation ("8 Viivi predefined procedures").
`transcript-on`	a procedure defined in R5RS to begin transcript output

[16]The copyright description

```
(viivi-set!  copyright "COPYRIGHT-STRING")
```

is now needed only in the configuration file in the Viivi system directory

`${VIIVI_HOME}viivi.conf`.

You no more have to write anything about the copyright in the user-editable configuration files (the other `viivi.conf`s or `.viivirc`s).

7 Viivi predefined constants

The following symbols are bound to the respective predefined constants in Viivi. These bindings are included in the base-level `32` (Viivi-specific predefined constants/procedures). When you want to use these symbols, not for the predefined constants, but as your own variables, you can simply override them using `define` or a procedure of the `let`-group. If you don't like to see the predefined constants, you can erase the predefined constans themselves from the predefined environment individually using the function `remove-predefined-symbol` of the procedure `viivi-set!` (See `viivi-set!` in "Section 8 Viivi predefined procedures").

7.1 Numerical constants

When they are evaluated in a mathematical expression, for example, they are immediately replaced with the respective predefined numerical values.

- `E`
  radix of natural logarithm
  `2.718281828459045`

- `PI`
  ratio of the circumference of a circle to its diameter
  `3.141592653589793`

7.2 Symbol constants

It is defined in order to help the Viivi novice.

- `help`
  This symbol is bound to another symbol over eleven-lines including empty-lines and carriage-returns, which summarizes information possibly helpful to the novice, such as how to quit the interpreter, etc. The bound value is not a string but a symbol, because if a string were used here, the quotation marks `" "` accompanied at the both ends of the string might confuse the novice.

```
(exit)                    to exit
[Control]+[D]             to exit in Unix systems
[Control]+[Z] [Enter]     to exit in Windows
(viivi-agreement)         to show agreement
(viivi-copyright)         to show copyright
(viivi-version)           to show version number
(viivi-title)             to show title
(viivi-help)              to show this help
help                      to show this help
```

8 Viivi predefined procedures

In addition to the procedures described in R5RS, the following procedures are predefined in Viivi.

- procedures to show information
  These procedures are always available independent of the base level configurations.

`(viivi-agreement)`	to show agreement
`(viivi-copyright)`	to show copyright
`(viivi-help)`	to show help messages
`(viivi-title)`	to show title
`(viivi-version)`	to show version
`(system-properties)`	to show all the properties of the platform

- procedures to obtain or to set properties of Viivi
  These procedures are always available independent of the base level configurations.

  `(viivi-property <VIIVI-PROPERTY-SYMBOL>)`
  to return the value of the property `<VIIVI-PROPERTY-SYMBOL>` of Viivi

  `(viivi-set! <VIIVI-PROPERTY-SYMBOL> [<VIIVI-PROPERTY-VALUE>])`
  to set the value `<VIIVI-PROPERTY-VALUE>` for the property `<VIIVI-PROPERTY-SYMBOL>` of Viivi

- procedures for random numbers
  These procedures are included in the base-level `32` (Viivi-specific predefined constants and procedures).

`(random [<N>])`	to return a pseudo-random value of floating point number or integer type (`<N>` is optional)
`(random-boolean)`	to return a pseudo-random boolean value
`(random-gaussian)`	to return a pseudo-random floating point value in the normal distribution of zero mean value and one standard deviation
`(random-seed [<N>])`	to specify a seed for the pseudo-random number generator (`<N>` is optional)

Each procedure will be detailed in the following.

8.1 Procedures to show information

Each procedure returns `#<unspecified>` after displaying information to the standard output port. These procedures were prepared to provide the user information. If you want to use the information in your program codes, use the procedure `viivi-property` instead.

- `(viivi-agreement)`

  Shows the agreement. Corresponds to the command line argument option `-agreement`.

- `(viivi-copyright)`

  Shows the copyright. Corresponds to the command line argument option `-copyright`.

- `(viivi-help)`

  Shows the help messages. Actually, this procedure displays the symbol value bound to a predefined symbol `help`, which is prepared in order to help the Viivi novice by letting her/him know how to quit the interpreter, etc.

- `(viivi-title)`

  Shows the title. Corresponds to the command line argument option `-title`.

- `(viivi-version)`

  Shows the version. Corresponds to the command line argument option `-version`.

- `(system-properties)`

  Shows all the information on the platform viewed from the JVM. Corresponds to the command line argument option `-sysprops`.

8.2 Procedures to obtain or to set properties of Viivi

These are procedures to obtain or to set the values of the properties of Viivi.

- `(viivi-property <VIIVI-PROPERTY-SYMBOL>)`

  Returns a property value of Viivi interpreter. It takes the symbol `<VIIVI-PROPERTY-SYMBOL>` for a property name or a special symbol as the argument. As the argument is evaluated, the symbol must have been quoted if it is directly given as the argument.
  This procedure was prepared mainly for the Scheme program codes to use the information. If the user wants to see the information, use the procedures listed above in "8.1. procedures to show information", instead.

  Example
  ```
  (viivi-property 'eval-argument-number)
  ```

  Each property name `<VIIVI-PROPERTY-SYMBOL>` and the respective function are listed below.

  - `base-level`

    Returns an integer value representing the current base-level, which is a bit-summation of the following elements:

1	A group of the objects defined in R5RS as `syntax` (and so forth).
2	`library-syntax`
4	`procedure`
8	`library-procedure`
16	`optional-procedure`
32	A group of Viivi-specific objects

    Note that this option must be used just for reference: Actual predefined symbols may have been appended/removed using `append-predefined-symbol` and/or `remove-predefined-symbol` on the current base-level. Use `predefined-symbols` to see the individual predefined symbols.

  - `continuation-argument-number`

    Returns an integer value (`0,1, or 2`) representing argument number accepted to continuation invocation. Viivi processes the continuation invocation with no arguments as a continuation invocation with an argument of `#<unspecified>`.

0	no arguments
1	one argument (R5RS compatible, default)
2	either no arguments or one argument

◦ `copyright`

Returns the string of the copyright description of Viivi.

◦ `debug-fold-multiple-columns`

Returns an integer value whose multiples of trace lines are folded and hidden in debug trace output.

◦ `debug-level`

Returns an integer value representing debug output level, which is a bit-summation of the following elements:

`0`	No debug trace outputs.
`1`	Outputs trace lines connecting an input S expression before evaluation and the value after evaluation.
`2`	Outputs evaluation counts.
`4`	Outputs generations and modifications of binding relations, and modifications of the internal structures of data.

◦ `debug-sleep-milliseconds`

Returns an integer value for which output stops in millisecond after every debug trace output (both for input S-expression and for its evaluated value).

◦ `direct-input-objects`

Returns the bit-summation of integer values which represent the kinds of object that can be input directly.

`0`	No direct input is accepted (default).
`1`	Direct input for the predefined objects (constants and procedures)

Examples

`#<constant:PI>`	Viivi-predefined constant `PI`
`#<procedure:cos>`	R5RS-predefined procedure `cos`

`2`	Direct input `#<nil>` for `()`
`4`	Direct input for `#<unspecified>` for `#<unspecified>`

◦ `eq?-comparison-mode`

Returns an integer value (`0, 1, 2` or `3`) representing `eq?` compares numbers and/or characters using whether their memory addresses or with their values.

0 `eq?` compares both the numbers and the characters using their memory addresses (default).
1 `eq?` compares the numbers using their values.
2 `eq?` compares the characters using their values.
3 `eq?` compares both the characters and the characters using their values.

◦ `eval-argument-number`

Returns an integer value (`1,2, or 3`) representing argument number accepted to `eval` invocation.

1 One argument (only the S-expression of the evaluation target and no environment specifier).
Example
```
(eval '(+ 20 30))
```
2 Two arguments (R5RS compatible, default).
Example
```
(eval '(+ 20 30) (scheme-report-environment 5))
```
3 Either one argument or two arguments.

◦ `flush-each-output`

Returns a boolean value (`#t` or `#f`) if an output port is flushed at every writing to the output port.

`#t` Flushes the output port every time.
`#f` Flushes the output not every time (depending on the platform).

◦ `help`

Shows available symbols to be used as `<VIIVI-PROPERTY-SYMBOL>`.
Returns `#<unspecified>`. Same as `usage`.

◦ `language`

Returns the character string representing the current language used in the interpreter.

`"en"` English
`"ja"` Japanese
`"de"` German

◦ `output-in-floating-point-number-format`

Returns a boolean value representing if the rational number value is output in the floating-point-number format (or otherwise in the rational-number format).

`#t`	The rational-number values are output in the floating-point-number format.
`#f`	The rational-number values are output in the rational-number format (default).

- `overwrite-output-files`

Returns a boolean value representing if output files are opened in overwrite-mode (or otherwise in append-mode). Output files are opened in the append-mode by default. Output to the network-ports is not affected by this setting. The command line argument options `-O` or `-o` can be also used, but the later designation takes precedence over the other earlier one.

`#t`	Output files are opened in overwrite-mode.
`#f`	Output files are opened in append-mode (default).

- `predefined-symbols`

Returns a list composed of all the symbols predefined on the current base level.

- `prompt`

Returns the character string currently used as Viivi's prompt.

- `return-value`

Returns an integer value (`0,1,2, or 3`) representing the values the procedures `define` and `set!` return.

`0`	Both `define` and `set!` return `#<unspecified>` (R5RS compatible, default).
`1`	Only `define` returns the newly bound value.
`2`	Only `set!` returns the newly bound value.
`3`	Both `define` and `set!` return the newly bound value.

- `system-properties`

Returns an association list composed of all the pairs of a property name string (as the key) and its property value string (as the datum) of the platform viewed from the JVM.

- `usage`

  Shows available symbols as `<VIIVI-PROPERTY-SYMBOL>`. Returns `#<unspecified>`. Same as `help`.

- `verbose-level`

  Returns an integer value (`0,1,2,3,4, or 5`) representing the output verbosity.

`0`	Does not output `#<unspecified>`
`1`	Does not output prompt
`2`	Outputs only error messages
`3`	Outputs also warning messages
`4`	Outputs title at the boot time
`5`	Outputs agreement at the boot time

- `version`

  Returns the string representing the version of Viivi.

- `(viivi-set! <VIIVI-PROPERTY-SYMBOL> [<VIIVI-PROPERTY-VALUE>])`

  Used to set properties of Viivi interpreter. The first argument `<VIIVI-PROPERTY-SYMBOL>` is a property symbol or a special symbol, and the second argument `<VIIVI-PROPERTY-VALUE>` is a property value. As the symbol of the first argument is not evaluated at the invocation, it must be provided without quotation. On the other hand, the second argument is evaluated at the invocation. Therefore, if a symbol is directly provided as the second argument, it must have been quoted.

  Example

  ```
  (viivi-set!  append-predefined-symbol 'cos)
  ```

  A few exceptions do not take the second argument. The function for each pair of the property symbol `<VIIVI-PROPERTY-SYMBOL>` and the corresponding property value `<VIIVI-PROPERTY-VALUE>` will be detailed in the following.

  - `append-predefined-symbol <PREDEFSYM>`

    The second argument `<PREDEFSYM>` is a symbol which is to be appended as a predefined symbol on the current base level. The symbol must be an appropriate one that is to be bound to an object prepared in the interpreter. When the symbol `<PREDEFSYM>` did not have a binding on the current base level, the symbol is newly bound to the corresponding object on the base level. When the symbol `<PREDEFSYM>` already had a binding on the current base level, it does nothing. There are no corresponding command line argument options at the present moment.

    Example

    ```
    (viivi-set!  append-predefined-symbol 'cos)
    ```

    When the symbol `cos` did not have a binding on the current base level to the procedure object `#<procedure:cos>` (defined in R5RS to obtain the value of a mathematical function cosine), then the symbol is newly bound to the procedure object on the current base level.

  - `continuation-argument-number <N>`

    Specifies an integer value `<N>` representing the number of the arguments accepted to continuation invocation.
    The second argument `<N>` is an integer `0,1, or 2`.
    This is equivalent to the command line argument option `-C <N>`.

`0`	No arguments.
`1`	One argument. (R5RS compatible, default)
`2`	Either zero arguments or one argument.

Example

`(viivi-set! continuation-argument-number 0)`
Specifies that the continuation invocation accepts no arguments.

◦ `copyright "COPYRIGHT"`

Specifies a character string `"COPYRIGHT"` representing the copyright. The interpreter confirms the copyright description. Don't delete or modify the copyright settings in the configuration files, or Viivi never boots up. When the copyright string was changed into a new one in a newly distributed package, then the copyright strings in all the configuration files need to be updated to the new copyright string.

◦ `debug-fold-multiple-columns <N>`

Specifies an integer value `<N>` whose multiples of trace lines are folded and hidden in debug trace output. The numbers of the folded and hidden trace lines are shown with digital numbers. This option is provided to display deep procedure invocations compact on the display with a limited width. This is equivalent to the command line argument option `-dc <N>`.

Example

`(viivi-set! fold-multiple-columns 20)`
Specifies that the trace lines multiples of 20 are folded and hidden.

◦ `debug-level <N>`

Specifies the level for debug trace output. The second argument `<N>` is a bit summation of `1`, `2`, and `4`. This is equivalent to the command line argument option `-d <N>`.

`0`	No debug trace outputs.
`1`	Outputs trace line connecting an input S-expression before evaluation and the value after evaluation.
`2`	Outputs evaluation count.
`4`	Outputs generations and modifications of binding relations, and modifications of the internal structures of data.

Example

`(viivi-set! debug-level 7)`
Turns Viivi into the debugging mode. This is equivalent to the command line argument option `-g`.

○ `debug-sleep-milliseconds <N>`

Specifies an integer value `<N>` for which debug trace output stops in millisecond at every output of input S-expression or its evaluated value. This is equivalent to the command line argument option `-ds <N>`.

○ `direct-input-objects <N>`

Specifies a bit summation `<N>` of integers which represent the kinds of object that can be input directly.

`0`	No direct input is accepted (default).
`1`	Direct input for the predefined objects (constants and procedures)

Example

`#<constant:PI>`	Viivi-predefined constant `PI`
`#<procedure:cos>`	R5RS-predefined procedure `cos`

`2`	Direct input `#<nil>` for the nil object `()`
`4`	Direct input `#<unspecified>` for the `#<unspecified>` object

○ `eq?-comparison-mode <N>`

Specifies an integer value (`<N> =0, 1, 2` or `3`) representing `eq?` compares numbers and/or characters whether with their memory addresses or with their values.

`0`	`eq?` compares both the numbers and the characters with their memory addresses (default).
`1`	`eq?` compares the numbers with their values.
`2`	`eq?` compares the characters with their values.
`3`	`eq?` compares both the characters and the characters with their values.

Example

```
> (eq?  2 2)
#f      ; eq? compares memory addresses of the numeric arguments by default.
> (viivi-set!  eq?-comparison-mode 1)
#<unspecified>
> (eq?  2 2)
#t      ; eq? was changed to compare values of the numeric arguments.
```

○ `eval-argument-number <N>`

Specifies the number of the arguments accepted to `eval` invocation. The second argument `<N>` is an integer value `1,2, or 3`. This is equivalent to the command line argument option `-E <N>`.

`1` One argument (only the S-expression of the evaluation target and no environment specifier)

Example

```
(eval '(+ 20 30))
```

`2` Two arguments (R5RS compatible, default)

Example

```
(eval '(+ 20 30) (scheme-report-environment 5))
```

`3` Either one argument or two arguments.

Example

```
(viivi-set!  eval-argument-number 1)
```

Specifies that the `eval` invocation accepts only one argument.

◦ `flush-each-output [#t|#f]`

Specifies if an output port is flushed at every writing to the output port. The second argument is a boolean value `#t` or `#f`. This is equivalent to the command line argument option `-u [true|false]`.

`#t` Flushes output ports at every output.
`#f` Does not flush output ports not at every output (depending on the platform).

Example

```
(viivi-set!  flush-each-output #t)
```

Specifies that the output ports flush at every output.

◦ `help`

Shows usage of `viivi-set!`. It does not take the second argument. Returns `#<unspecified>`. Same as `usage`.

◦ `language ["en"|"ja"|"de"]`

Specifies the (natural) language that the interpreter uses. Currently, English, Japanese, and German are supported. The argument is one of the character strings representing these languages.

`"en"` English
`"ja"` Japanese
`"de"` German

If the specified language is not available, English (C locale) will be used. Note that characters may be possibly garbled in case of specifying a language different from the current

locale of the platform. This is equivalent to the command line argument option `-LANG [en|ja|de]`.

- `output-in-floating-point-number-format [#t|#f]`

  Specifies a boolean value representing if the rational number value is output in the floating-point-number format (or otherwise in the rational-number format). Equivalent to the command line argument option `-F [true|false]`.

  - `#t` The rational-number values are output in the floating-point-number format.
  - `#f` The rational-number values are output in the rational-number format (default).

- `overwrite-output-files [#t|#f]`

  Specifies a boolean value representing if the output files are opened in overwrite-mode (or otherwise, in append-mode). Output files are opened in the append-mode by default. Output to the network-ports is not affected by this setting. The command line argument options `-O` or `-o` can be also used, but the later designation takes precedence over the other earlier one.

  - `#t` Output files are opened in overwrite-mode.
  - `#f` Output files are opened in append-mode (default).

- `pop-base-level`

  Discards the current base level and returns to the previous base level. It does not take the second argument. When the current base level was the first one generated at the boot time, it does nothing. This is equivalent to the command line argument option `-Bx`.

- `prompt "PROMPT"`

  Specifies Viivi's prompt. The second argument `"PROMPT"` is a character string used for Viivi's prompt. This is equivalent to the command line argument option `-p PROMPT`. (But this procedure invocation is recommended, because the command line argument may be expanded by shells to an unexpected character string.)

  Example

  ```
  (viivi-set!  prompt "[Experiment-1]")
  ```
  Change Viivi's prompt to `[Experiment-1]`.

◦ `push-base-level <N>`

Generates a new base level and sets it as the current base level. It takes as the second argument `<N>` an integer value of a bit summation of 2^n `(n=0,1,...,5)`. This is equivalent to the command line argument option `-B <N>`. Reversion to the previous base level can be performed by a procedure invocation `(viivi-set! pop-base-level)` or by a command line argument option `-Bx`.

1	A group of the objects defined in R5RS as `syntax` (and so forth).
2	`library-syntax`
4	`procedure`
8	`library-procedure`
16	`optional-procedure`
32	A group of Viivi-specific objects

Example

```
(viivi-set! push-base-level 5)
```

Generates a new base level only with the objects defined in R5RS as `syntax` and `procedure` and sets it as the current base level.

◦ `remove-predefined-symbol <PREDEFSYM>`

The second argument `<PREDEFSYM>` is a symbol which is to be removed from the current base level. The symbol must be an appropriate one that is to have been bound to an object prepared in the interpreter. When the symbol `<PREDEFSYM>` had a binding on the current base level, the binding is removed from the base level. When the symbol `<PREDEFSYM>` did not have a binding on the current base level, it does nothing. There are no corresponding command line argument options at the present moment.

Example

```
(viivi-set! append-predefined-symbol 'cos)
```

When the symbol `cos` had a binding on the current base level to the procedure object `#<procedure:cos>` (defined in R5RS to obtain the value of a mathematical function cosine), then the binding is removed from the current base level.

◦ `return-value <N>`

Specifies the values returned by the procedures `define` and `set!`. The second argument `<N>` is an integer value (`0,1,2,or 3`) representing the values `define` and `set!` return. This is equivalent to the command line argument option `-R <N>`.

0 Both `define` and `set!` return `#<unspecified>` (R5RS compatible, default).
1 Only `define` returns the newly bound value.
2 Only `set!` returns the newly bound value.
3 Both `define` and `set!` return the newly bound values.

Example

```
(viivi-set!  return-value 2)
```

Specifies that the procedure `set!` returns the newly bound value. The procedure `define` returns `#<unspecified>`.

◦ `usage`

Shows usage of `viivi-set!`. It does not take the second argument. Returns `#<unspecified>`. Same as `help`.

◦ `verbose-level <N>`

Specifies verbosity for the output. The second argument `<N>` is an integer value (`0,1,2,3,4, or 5`) representing the output verbosity. This is equivalent to the command line argument option `-v <N>`.

0 Does not output `#<unspecified>`.
1 Does not output prompt.
2 Outputs only error messages.
3 Outputs also warning messages.
4 Outputs title at the boot time.
5 Outputs agreement at the boot time.

Example

```
(viivi-set!  verbose-level 2)
```

Specifies a silent verbosity for the output (suppressing warning messages).

8.3 Procedures for random numbers

These procedures are implemented especially to write game programs! Since they are not defined in R5RS, they are currently implemented in the style of Java API. The style can be changed in the future versions according to new Scheme standards.

- `(random [<N>])`

  Being invoked with the argument `<N>` of a positive integer value, it returns a pseudo-random integer i such as $0 \leq i < $ `<N>`. Being invoked with no arguments, it returns a pseudo-random floating point number r such as $0 \leq r < 1.0$.
  Examples

  `(random 100)`
  Returns a pseudo-random integer value i such as $0 \leq i < 99$.

  `(* 5 (random))`
  Returns a pseudo-random floating point number value r such as $0 \leq r < 5.0$.

- `(random-boolean)`

  Returns a pseudo-random boolean value of `#t` or `#f`.

- `(random-gaussian)`

  Returns a pseudo-random value of floating point number in the normal distribution of zero mean value and one standard deviation.

- `(random-seed [<N>])`

  When it is invoked with a 64-bit integer argument value `<N>`, it uses `<N>` as a seed for the pseudo-random number generation. When it is invoked without arguments, a 64-bit integer value converted from the clock time at the invocation is used as a seed for the pseudo-random number generation. This procedure is invoked once before using other random number procedures, if the user explicitly initializes the random-number generator. When this procedure is not invoked explicitly, the initialization at the boot time is used.

  Examples

  `(random-seed 16741)`
  Integer value `16741` will be used as a random seed.

  `(random-seed)`
  An integer converted from the invocation clock time will be used as a random seed.

9 Viivi command line argument options

After processing all the configuration files, Viivi usually processes the command line argument options sequentially from left to right. Their order of appearance is therefore meaningful.

9.1 Command line argument options to show information

Each of the following options shows corresponding information to the standard output port and immediately quits Viivi. Note that they are the only exceptions for the rule of 'left-to-right' sequential processing on the command line arguments: When a command line argument option of this group was found anywhere in the command line argument option sequence, then Viivi quits immediately after processing it, with all the other options ignored.

- `-agreement`
  Shows the agreement. Corresponds to the procedure invocation `(viivi-agreement)`.

- `-copyright`
  Shows the copyright. Corresponds to the procedure invocation `(viivi-copyright)`.

- `-flags`
  Shows a list of available command line argument options. Same as `-h[elp]`.

- `-h[elp]`
  Shows a list of available command line argument options. Same as `-flags`.

- `-sysprops`
  Shows all the information on the platform viewed from the JVM. Corresponds to the procedure invocation `(system-properties)`.

- `-title`
  Shows the title. Corresponds to the procedure invocation `(viivi-title)`.

- `-ver[sion]`
  Shows the version. Corresponds to the procedure invocation `(viivi-version)`.

9.2 Command line argument options for R5RS incompatibilities

Viivi accepts by default only Scheme codes compatible with R5RS. On the other hand, however, the user can also let Viivi accept Scheme codes incompatible with R5RS that are already existing everywhere in the world, by providing specific command line argument options.

- `-C <N>`

  Specifies an integer value `<N>` representing the number of the arguments accepted to continuation invocation. The argument `<N>` is an integer `0,1, or 2`. This is equivalent to the procedure invocation

  `(viivi-set! continuation-argument-number <N>)`.

`0`	No arguments.
`1`	One argument. (R5RS compatible, default)
`2`	Either zero arguments or one argument.

  Example

  `$ viivi -C 0`
  Specifies that the continuation invocation accepts no arguments.

- `-E <N>`

  Specifies the number of the arguments accepted to `eval` invocation.
  The argument `<N>` is an integer value `1,2, or 3`.
  This is equivalent to the procedure invocation

  `(viivi-set! eval-argument-number <N>)`.

`1`	One argument.
`2`	Two arguments (R5RS compatible, default).
`3`	Either one argument and two arguments.

  Example

  `$ viivi -E 1`
  Specifies that the `eval` invocation accepts only one argument.

- `-Q <N>`
  Specifies an integer value `<N>` (`0, 1, 2` or `3`) representing `eq?` compares numbers/characters whether with their memory addresses or with their values.

0	`eq?` compares both numbers and characters with their memory addresses (default).
1	`eq?` compares numbers with their values.
2	`eq?` compares characters with their values.
3	`eq?` compares both numbers and characters with their values.

Example

```
$ viivi -Q 1
```

Changes `eq?` to compare values of the numeric arguments.

- `-R <N>`

  Specifies the values returned by `define` and `set!`. The second argument `<N>` is an integer value (`0,1,2,or 3`) representing the values `define` and `set!` return. This is equivalent to the procedure invocation `(viivi-set! return-value <N>)`.

0	Both `define` and `set!` return `#<unspecified>` (R5RS compatible, default).
1	Only `define` returns the newly bound value.
2	Only `set!` returns the newly bound value.
3	Both `define` and `set!` return the newly bound values.

Example

```
$ viivi -R 2
```

Specifies that only the procedure `set!` returns the newly bound value. The procedure `define` returns `#<unspecified>`.

9.3 Command line argument options for specifying Viivi's behaviors

The following command line argument options specify to change behaviors in evaluation and input/output of Viivi.

- `-B <N>`

  Generates a new base level and sets it as the current base level. The argument parameter `<N>` is a bit summation of integer values 2^n `(n=0,1,...,5)` representing the groups of predefined objects. This is equivalent to the procedure invocation `(viivi-set! push-base-level <N>)`. Reversion to the previous base level can be performed by a procedure invocation `(viivi-set! pop-base-level)` or by a command line argument option `-Bx`.

1	A group of the objects defined in R5RS as `syntax` (and so forth).
2	`library-syntax`
4	`procedure`
8	`library-procedure`
16	`optional-procedure`
32	A group of Viivi-specific objects

  Example

  ```
  $ viivi -B 5
  ```
  Generates a new base level only with the objects defined in R5RS as `syntax` (represented with `1`) and `procedure` (represented with `4`) and uses it as the current base level.

- `-Bx`

  Discards the current base level and returns to the previous base level. Takes no second argument. When the current base level was the first one generated at the boot time, it does nothing. This is equivalent to the procedure invocation `(viivi-set! pop-base-level)`.

  Example

  ```
  $ viivi -B 5 -i test1.scm -B 31 -i test2.scm -Bx -i test3.scm
  ```
  Firstly, generating a new base level (say, B5) of ensemble of the predefined object groups defined in R5RS as `syntax` (represented with `1`) and `procedure` (represented with `4`) and setting it as a current base level, input from `test1.scm` is evaluated. And next, generating a new base level (say, B31) of all the object groups predefined in R5RS (`31=1+2+4+8+16`) and setting it as a current base level, input from `test2.scm` is evaluated. Finally, after discarding the latest base level (B31) and getting back to the previous base level (B5), input from `test3.scm` is evaluated.

- `-D <N>`

  Specifies the bit-summation `<N>` of integer values (`1`,`2`, or `4`) representing the kinds of object

that can be input directly.

`0`	No direct input is accepted (default).
`1`	Direct input for the predefined objects (constants and procedures)

Examples

`#<constant:PI>`	Viivi-predefined constant `PI`
`#<procedure:cos>`	R5RS-predefined procedure `cos`

`2`	Direct input `#<nil>` for the nil object `()`
`4`	Direct input `#<unspecified>` for the `#<unspecified>` object

- `-d <N>`

Specifies the level for debug trace output. The second argument `<N>` is a bit summation of `1`, `2`, and `4`. This is equivalent to the procedure invocation `(viivi-set! debug-level <N>)`.

`0`	No debug trace outputs (default).
`1`	Outputs trace line connecting an input S-expression before evaluation and the value after evaluation.
`2`	Outputs evaluation count.
`4`	Outputs generations and modifications of binding relations, and modifications in structures of data.

Example

```
$ viivi -d 7
```

This turns Viivi into the debugging mode. Equivalent to the command line argument option `-g` or to the procedure invocation `(viivi-set! debug-level 7)`.

- `-dc <N>`

Specifies an integer value `<N>` whose multiples of trace lines are folded and hidden in debugging mode. The number of the folded and hidden trace lines are shown with digital numbers. The argument value `<N>` is an arbitrary integer value. Equivalent to the procedure invocation `(viivi-set! debug-fold-multiple-columns <N>)`.

Example

```
$ viivi -g -dc 20
```

Specifies that the trace lines multiples of 20 are folded and hidden .

- `-ds <N>`

Specifies an integer value in millisecond `<N>` for which every debug trace output of input S-expression or its evaluated value stops. Equivalent to the procedure invocation `debug-sleep-milliseconds <N>)`.

Example

```
$ viivi -g -ds 3000
```
Waits 3 seconds after every output in debugging mode.

- `-F [true|false]`

  Specifies if the rational number values are output in the floating-point-number format (or otherwise in the rational-number format). Equivalent to the procedure invocation `(viivi-set! output-in-floating-point-number-format [#t|#f])`.

`true`	Output values of rational numbers in the floating-point-number format
`false`	Output values of rational numbers in the rational number format (default)

- `-g`

  Sets up a debugging mode in a handy way. Equivalent to the command line argument option `-d 7` or to the procedure invocation `(viivi-set! debug-level 7)`.

- `-I [true|false]`

  Specifies if an error in opening a input port given on the command line is ignored and the process is continued. The argument value is either `true` or `false`. There are no equivalent procedure invocations.

`true`	Ignores the opening port error and proceeds.
`false`	Reports an error (default).

- `-LANG [en|ja|de]`

  Specifies the (natural) language that the interpreter uses. Currently, English, Japanese, and German are supported. The argument is one of the character strings representing these languages.

`en`	English
`ja`	Japanese
`de`	German

  If the specified language is not available, English (C locale) will be used. Note that characters may be possibly garbled in case of specifying a language different from the current locale of the platform. equivalent to the procedure invocation `(viivi-set! language ["en"|"ja"|"de"])`.

- `-p "PROMPT"`

  Specifies Viivi's prompt. The argument `"PROMPT"` is a character string used for Viivi's prompt. This is equivalent to the procedure invocation `(viivi-set! prompt "PROMPT")`. But the procedure invocation is recommended, because the command line argument may be expanded by shells to an unexpected character string (such that spaces may be dropped).

Example

```
$ viivi -p "[Experiment-1]"
```
Changes Viivi's prompt to `[Experiment-1]`.

- `-u [true|false]`

  Specifies if an output port is flushed at every writing to the output port. The argument is a boolean value `true` or `false`.
  This is equivalent to the procedure invocation `(viivi-set! flush-each-output [#t|#f])`.

`true`	Flushes output ports at every output (default).
`false`	Does not flush output ports at every output (depending on the platform).

  Example

  ```
  $ viivi -u true
  ```
  Specifies that the output ports flush at every output.

- `-v <N>`

  Specifies verbosity for the output. The argument `<N>` is an integer value (`0,1,2,3,4, or 5`) representing the output verbosity. This is equivalent to the procedure invocation `(viivi-set! verbose-level <N>)`.

`0`	Does not output `#<unspecified>`.
`1`	Does not output prompt.
`2`	Outputs only error messages.
`3`	Outputs also warning messages.
`4`	Outputs title at the boot time.
`5`	Outputs agreement at the boot time.

  Example

  ```
  $ viivi -v 2
  ```
  Specifies a silent verbosity for the output (suppressing warning messages).

- `-W [true|false]`

  Specifies a boolean value representing if the output files are opened in overwrite-mode (or otherwise, in append-mode). Output files are opened in the append-mode by default. Output to the network-ports is not affected by this setting. The procedure invocation `(viivi-set! overwrite-output-files [#t|#f])` can be also used, but the later designation takes precedence over the other earlier one.

`true`	Output files are opened in overwrite-mode.
`false`	Output files are opened in append-mode (default).

9.4 Command line argument options for input-source/output-destination

The following command line argument options specify input-source/output-destination.
Here special symbols are used for the explanation.

`FILE`	a filename string
`HOST`	a string representing a hostname or an IP address of a host
`:PORT`	a network-port number following to a colon character `':'`

- `-e [FILE|(HOST):PORT]`

  Changes error output destination to the specified file or network-port.

  Examples

  ```
  $ viivi -e errorOutput.txt
  ```
  After changing error output destination to the file `errorOutput.txt`, input from the standard input port is evaluated.

  ```
  $ viivi -e /dev/stdout test.scm
  ```
  After changing error output destination to the file `/dev/stdout` (actually the standard output port on Unix-like platforms), input from the file `test.scm` is evaluated.

  ```
  $ viivi -e suuritammi:5963
  ```
  After changing error output destination to the network-port `5963` on the host named `suuritammi`, input from the standard input port is evaluated.

  ```
  $ viivi -e 192.168.20.51:5963
  ```
  After changing error output destination to the network-port `5963` on the host of IP-address `192.168.20.51`, input from the standard input port is evaluated.

  ```
  $ viivi -e :11922
  ```
  After changing error output destination to the network-port `11922` of the `localhost`, input from the standard input port is evaluated. Same as `$ viivi -e localhost:11922`.

- `-f [FILE|:PORT]`

  After evaluating input from the specified file or network-port, input from the standard input is evaluated. If multiple input sources are provided on a command line (whether with or without flags `-f`, `-i`, or `-L`) and at least one of them is specified with the `-f` flag, input from all the input sources is evaluated sequentially, and finally input from the standard input port is evaluated.

  Examples

  ```
  $ viivi -f ../scheme/first.scm
  ```
  After evaluating input from the file `../scheme/first.scm`, input from the standard input port is evaluated.

`$ viivi -f prepare.scm -i test1.scm`
Evaluating input from the file `prepare.scm` first, input from test1.scm is evaluated next, and finally input from the standard input port is evaluated Same as `$ viivi -f prepare.scm test1.scm`.

`$ viivi -f :11291`
After evaluating input from the network-port 11291 of the `localhost`, input from the standard input port is evaluated.

- `-i [FILE|:PORT]`

  Evaluates input from the specified file or network-port.

  Examples

  `$ viivi -i source.scm`
  Input from the file `source.scm` is evaluated.
  Same as `$ viivi source.scm`

  `$ viivi -i :46497`
  Input from the network-port `46497` of the `localhost`.
  Same as `$ viivi localhost:46497` or `$ viivi :46497`.

- `-L [FILE|:PORT]`

  After performing "library input" from the specified file or network-port, input from the following input sources is evaluated. If no following input sources are specified, input from the standard input port is evaluated. "Library input" is an input in which side effects of evaluation are effective but no output is performed. The purpose of this option is to provide a method for reading beforehand as a library a file/network-port of reliable Scheme codes that have been tested enough. Users are interested only in the side effects, but not in the output, of the library codes. This option can be used also to read a normal Scheme file as a virtual configuration file with the highest priority. If you need output during the evaluation as well, use `-f` option, instead.

  Examples

  `$ viivi -L /usr/local/lib/scheme/math.scm -i model.scm`
  After library-inputting from the file `/usr/local/lib/scheme/math.scm`, input from the file `model.scm` is evaluated. Same as `$ viivi -L /usr/local/lib/scheme/math.scm model.scm`.

  `$ viivi -L :49894`
  After library-inputting from the network-port `49894` of the `localhost`, input from the standard input port of the platform is evaluated. Same as `$ viivi -L localhost:49894`.

- `-O [FILE|(HOST):PORT]`

  Changes standard output destination to the specified file or network-port. Note that the contents of the specified file will be overwritten, when the file already exists. To keep the contents of the existing file and to append newly output contents to the end of the file, use `-o` option, instead.

  Examples

  `$ viivi -O output.result`
  After changing standard output destination to the file `output.result`, input from the standard input port is evaluated. When the file `output.result` already existed, its contents are overwritten.

  `$ viivi -O kotkanpoika:42809`
  After changing standard output destination to the network-port `42809` of the host `kotkanpoika`, input from the standard input port is evaluated.
  Same as `$ viivi -o kotkanpoika:42809`.

  `$ viivi foo.scm -O bar.data baz.scm`
  Firstly, input from the file `foo.scm` is evaluated. (The standard output destination is still the standard output port of the platform.) Next, the standard output destination is changed to the file `bar.data`, and finally input from the file `baz.scm` is evaluated. When the file `bar.data` already existed, its contents are overwritten.

- `-o [FILE|(HOST):PORT]`

  Changes the standard output destination to the specified file or network-port. Note that the new output contents are appended to the end of the old contents in the file, when the specified file already existed. To overwrite the contents of the existing file with the standard output contents, use `-O` option, instead.

  Examples

  `$ viivi -o output.result`
  After changing the standard output destination to the file `output.result`, input from the standard input port is evaluated. When the file `output.result` already existed, the output contents are appended to the end of the file.

  `$ viivi -o 192.168.12.34:56789`
  After changing the standard output destination to the network-port `56789` of the host with IP-address `192.168.12.34`, input from the standard input port is evaluated.
  Same as `$ viivi -O 192.168.12.34:56789`.

  `$ viivi -o dst1.txt -i src1.scm -o dst2.txt -i src2.scm`
  After changing the standard output destination to the file `dst1.txt`, input from the file `src1.scm` is evaluated. And then changing the standard output destination to the file `dst2.txt`, input from the file `src2.scm` is evaluated. When the files `dst1.txt` or `dst1.txt` already existed, the

output contents are appended to the ends of the files, respectively. Consequently, the results of the evaluations from the source files `src1.scm` and `src2.scm` are stored into the destination files `dst1.txt` and `dst2.txt`, respectively.

(The next sample is a possible mistake that does not work in the way you would expect:
`$ viivi -i src1.scm -o dst1.txt -i src2.scm -o dst2.txt  # WRONG!`
The evaluation results from the source file `src1.scm` will be written to the default standard output port, and the evaluation results from the source file `src2.scm` will be written to the destination file `dst1.txt`. Nothing is written to the destination file `dst2.txt`.)

- `-t [FILE|(HOST):PORT]`

  Sets transcript output destination to the specified file or network-port. When the file already exists, output contents will be always appended to the end of the file. When the transcript output is already performed to another output destination, the output destination is closed and then the specified output destination is newly opened. Equivalent to the procedure invocation `(transcript-on "[FILE|(HOST):PORT]")`.

  Example

  `$ viivi -t carbonCopy.log`
  After setting transcript output destination to the file `carbonCopy.log`, input from the standard input port is evaluated. When the file `carbonCopy.log` already existed, contents of the transcript output are appended to the end of the file.

- `-tx`

  Closes the current transcript output port. When a transcript output port was not open, it does nothing. Equivalent to the procedure invocation `(transcript-off)`.

- `FILE(S)`
  `:PORT(S)`

  Reads and evaluates the input from the files and/or network-ports sequentially in the order as they are specified. Files and network-ports can be mixed in an arbitrary order. Equivalent to the command line argument options in which the option `-i` is inserted in front of each filename and network-port. Regardless whether single or multiple, filename(s) and network-port(s) without option switches must be located at the end of the command line.

  Examples

  `$ viivi Heidi.scm Klara.scm Peter.scm`
  Reads and evaluates the input from the files `Heidi.scm`, `Klara.scm`, and `Peter.scm`, sequentially. Equivalent to the command line argument options
  `$ viivi -i Heidi.scm -i Klara.scm -i Peter.scm`

`$ viivi :17771 Paul.scm :17772 Elizabeth.scm`
Reads and evaluates the input from the network-port `17771` of the `localhost`, the file `Paul.scm`, the network-port `17772` of the `localhost`, and the file `Elizabeth.scm`, sequentially. Equivalent to the command line argument options
`$ viivi -i :17771 -i Paul.scm -i :17772 -i Elizabeth.scm`

10 R5RS specifications of Viivi

10.1 Equality testing procedures of Viivi

R5RS allows a little variance in the implementations for `eq?`/`eqv?`/`equal?`.
Here summarize the behaviors of the equality testing procedures in Viivi.

Table 1: Summary of equality testing procedures

	`eq?`	`eqv?`	`equal?`	
nil (empty list)	type	type	type	
unspecified	type	type	type	
EOF (End Of File)	type	type	type	
symbol	string expression	string expression	string expression	
boolean	value	value	value	
number	address / [value+exactness]	[value+exactness]	[value+exactness]	`eq?` is switchable
character	address / value	value	value	`eq?` is switchable
string	address	value	value	
vector	address	address	recursive test	
pair	address	address	recursive test	
OTHERS	address	address	address	

switchable for number and character*

type:	`#t` if the types of the argument objects are the same; `#f` otherwise
string expression:	`#t` if the string expressions are the same; `#f` otherwise (only for symbols)
value:	`#t` if the values are the same; `#f` otherwise
address:	`#t` if the memory addresses of the argument objects are the same; `#f` otherwise
[value+exactness]:	`#t` if both the values and the exactness are the same; `#f` otherwise (only for numbers)
recursive test:	compare recursively applying `equal?`

* Behavior of `eq?` comparing numbers and/or characters can be switched in Viivi. There are two ways to switch:

- procedure invocation `(viivi-set! eq?-comparison-mode <N>)`
- command line argument `-Q <N>`

The way of comparison tests by `eq?` depends on the value of `<N>` as follows:

`0`	`eq?` compares both the numbers and the characters with their memory addresses (default).
`1`	`eq?` compares the numbers with their values.
`2`	`eq?` compares the characters with their values.
`3`	`eq?` compares both the characters and the characters with their values.

10.2 General timing of argument evaluation in a procedure invocation

Invoking of a procedure is performed in the following order:

1. All the arguments (to be evaluated) given to the procedure are evaluated from left to right.
2. The own function of the procedure is processed.

11 Unique extensions of Viivi

The extensions introduced in this section are uniquely implemented in Viivi and are not defined in R5RS. Remark that the Scheme codes using these extensions do NOT basically run on other implementations,

11.1 Viivi debug trace mechanism

Viivi is equipped with a unique feature of debug trace mechanism which visualizes the evaluating processes in a natural way. It shows lines connecting input S-expressions and the corresponding evaluated values.
It does not show those for the S-expressions not to be evaluated (e.g., the first symbol for `define` or `set!`, the symbol for the name of a named-`let`, or the first list of the argument for `lambda`, and etc.).

To enable the debug trace function, do one of the following:

- to give a command line argument option `-g` or `-d <N>`
- to invoke a procedure `(viivi-set! debug-level <N>)`

The parameter `<N>` takes a bit-summation of integer values of `1`,`2`, and `4`, each of which means:

`0` No debug trace outputs (default).

`1` Outputs trace lines connecting an input S expression before evaluation and the value after evaluation.

`2` Outputs evaluation counts.

`4` Outputs generations and modifications of binding relations, and modifications of the internal structures of data.
Each of the outputs of `4` is distinguished with its own marks:

- ○ `symbol` → `valueA` generation of a new binding
- • `symbol` → `valueB` modification of an existing binding
- ■ `datumC` ⇒ `datumD` modification in the structure of an existing datum

Usually each trace line is closed connecting an input S-expression and its evaluated value as shown in Example 1 below.

Example 1

Try to get into the debugging mode and then to input `30` and `(+ 20 30)`.
(Some parts were edited for explanation.)

```
$ viivi
> (viivi-set!  debug-level 7)       ; Getting into debugging mode using viivi-set!.
#<unspecified>
> 30                                ; Inputting 30.

      0: ┌── 30                      ; Input S-expression 30 before evaluation.
      1: └── 30                      ; Integer value 30 after evaluation. Evaluation count increased from 0 to 1.
30                                   ; 30 was returned as a result.
> (+ 20 30)                         ; Inputting (+ 20 30).

      1: ┌── (+ 20 30)               ; Input S-expression (+ 20 30) before evaluation.
      1: │  ┌── +                    ; The first element S-expression + in the list.
      2: │  └── #<procedure:+>       ; A procedure #<procedure:+> was returned after evaluation.
      2: │  ┌── 20                   ; The second element S-expression 20 in the list.
      3: │  └── 20                   ; An integer value 20 was returned after evaluation.
      3: │  ┌── 30                   ; The third element S-expression 30 in the list.
      4: │  └── 30                   ; An integer value 30 was returned after evaluation.
      5: └── 50                      ; The procedure #<procedure:+> was applied to 20 and 30
50                                   ; to give the final result 50.
> (exit)                            ; Quitting with (exit).

      5: ┌── (exit)                  ; Input S-expression (exit).
      5: │  ┌── exit                 ; The first S-expression exit.
      6: │  └── #<procedure:exit>    ; A procedure #<procedure:exit> was returned after evaluation.
/// Interpreter terminated normally.    ; The procedure #<procedure:exit> was invoked to exit.
$
```

Example 2

Next is a little advanced example. The following shows the debug trace outputs for the definition and an invocation of a closure `distance-from-origin` which returns the distance $r = \sqrt{x^2 + y^2}$ from the origin to a point whose x- and y-coordinates are given as the arguments.

```
> (define distance-from-origin                          ; Bind to the symbol distance-from-origin
          (lambda (x y) (sqrt (+ (* x x) (* y y)))))    ; the closure created by the lambda invocation.

   0: ┌── (define distance-from-origin (lambda (x y) (sqrt (+ (* x x) (* y y)))))
   0: │ ┌── define
   1: │ └── #<procedure:define>                ; an R5RS-predefined procedure to generate a new binding
   1: │ ┌── (lambda (x y) (sqrt (+ (* x x) (* y y))))
   1: │ │ ┌── lambda
   2: │ │ └── #<procedure:lambda>              ; an R5RS-predefined procedure to create a closure
   3: │ └── #<closure:  [(x y)] [(sqrt (+ (* x x) (* y y)))]>                          ; a new closure
   3: │ ○ distance-from-origin → #<closure:  [(x y)] [(sqrt (+ (* x x) (* y y)))]>     ; a new binding
   4: └── #<unspecified>
#<unspecified>
> (distance-from-origin 7 8)                    ; To obtain the distance from the origin to the point (7,8).

   4: ┌── (distance-from-origin 7 8)
   4: │ ┌── distance-from-origin
   5: │ └── #<closure:  [(x y)] [(sqrt (+ (* x x) (* y y)))]>
   5: │ ┌── 7
   6: │ └── 7
   6: │ ┌── 8
   7: │ └── 8
   7: │ ○ x → 7                               ; New bindings x → 7 and y → 8 were
   7: │ ○ y → 8                               ; generated as the arguments for the closure.
   7: │ ┌── (sqrt (+ (* x x) (* y y)))
   7: │ │ ┌── sqrt
   8: │ │ └── #<procedure:sqrt>                ; an R5RS-predefined procedure to calculate the square-root
   8: │ │ ┌── (+ (* x x) (* y y))
   8: │ │ │ ┌── +
   9: │ │ │ └── #<procedure:+>
   9: │ │ │ ┌── (* x x)
   9: │ │ │ │ ┌── *
  10: │ │ │ │ └── #<procedure:*>
  10: │ │ │ │ ┌── x
  11: │ │ │ │ └── 7
  11: │ │ │ │ ┌── x
  12: │ │ │ │ └── 7
  13: │ │ │ └── 49                              ; the evaluation result for (* 7 7)
  13: │ │ │ ┌── (* y y)
  13: │ │ │ │ ┌── *
  14: │ │ │ │ └── #<procedure:*>
  14: │ │ │ │ ┌── y
  15: │ │ │ │ └── 8
  15: │ │ │ │ ┌── y
  16: │ │ │ │ └── 8
  17: │ │ │ └── 64                              ; the evaluation result for (* 8 8)
  18: │ │ └── 113                               ; the evaluation result for (+ (* 7 7) (* 8 8))
  18: │ └── 10.63014581273465                   ; the evaluation result for (sqrt (+ (* 7 7) (* 8 8)))
  19: └── 10.63014581273465                     ; Floating-point number calculation in Viivi is
10.63014581273465                               ; always performed with a 64-bit precision.
>
```

The following Example 3 and Example 4 are the exceptions showing the trace lines with their ends not connected to S-expressions or their evaluated values.

Example 3

The sample codes on `<http://en.wikipedia.org/wiki/Continuation>` were written in the file `Continuation.scm`, and then were fed to Viivi with the debugging flag `-g`.

Remark:

- As the codes invoke continuations with no arguments, you need to make beforehand Viivi accept the continuation-invocation style using either of the following:
  - the command-line argument option `-C 0` (Here we use this.)
  - the procedure invocation `(viivi-set!  continuation-argument-number 0)`
- It was partly edited at long lines and at the parts to be explained.

```
$ viivi -C 0 -g Continuation.scm
>
   0: ┌── (define the-continuation #f)
   0: │ ┌── define
   1: │ └── #<procedure:define>
   1: │ ┌── #f
   2: │ └── #f
   2: │ ○ the-continuation → #f     ; The symbol the-continuation was newly bound (initialized) to #f.
   3: └── #<unspecified>
#<unspecified>
>
   3: ┌── (define (test)     ; When the first argument for define is a list, lambda is implicitly invoked to define a closure.
   3: │     (let ((i 0))
   3: │       (call/cc (lambda (k) (set!  the-continuation k)))
   3: │       (set!  i (+ i 1))
   3: │       i))
   3: │ ┌── define
   4: │ └── #<procedure:define>
   4: │ ┌── (#<procedure:implicit-lambda> ()        ; lambda is implicitly invoked.
   4: │ │     (let ((i 0))
   4: │ │       (call/cc (lambda (k) (set!  the-continuation k)))
   4: │ │       (set!  i (+ i 1))
   4: │ │       i))
   4: │ └── #<closure:  []                          ; A closure was created.
   4: │       [(let ((i 0))
   4: │         (call/cc (lambda (k) (set!  the-continuation k)))>
   4: │         (set!  i (+ i 1))
   4: │         i)]>
   4: │ ○ test → #<closure:  []                     ; The symbol test was initialized to the closure.
   4: │             [(let ((i 0))
   4: │               (call/cc (lambda (k) (set!  the-continuation k)))
   4: │               (set!  i (+ i 1))
   4: │               i)]>
   5: └── #<unspecified>
#<unspecified>
>
   5: ┌── (test)                                    ; To invoke test.
   5: │ ┌── test
   6: │ └── #<closure:  []
   6: │       [(let ((i 0))
   6: │         [(call/cc (lambda (k) (set!  the-continuation k)))
   6: │         [(set!  i (+ i 1))
   6: │         [i)]>
   6: │ ┌── (let ((i 0))
   6: │ │     (call/cc (lambda (k) (set!  the-continuation k)))
   6: │ │     (set!  i (+ i 1)))
   6: │ │     i)
   6: │ │ ┌── let
   7: │ │ └── #<procedure:let>
```

```
  7: │ │ ┌── 0
  8: │ │ └── 0
  8: │ │ ○ i → 0                                          ; The symbol i was initialized to 0.
  8: │ │ ┌── (call/cc (lambda (k) (set!  the-continuation k)))
  8: │ │ │ ┌── call/cc
  9: │ │ │ └── #<procedure:call-with-current-continuation>
  9: │ │ │ ┌── (lambda (k) (set!  the-continuation k))
  9: │ │ │ │ ┌── lambda
 10: │ │ │ │ └── #<procedure:lambda>
 11: │ │ │ └── #<closure:  [(k)] [(set!  the-continuation k)]>
 11: │ │ │ │ ○ k → #<continuation:0>    ; The symbol k was initialized to the continuation (with the binding i → 0).
 11: │ │ │ │ ┌── (set!  the-continuation k)                                                    ;<--[B]
 11: │ │ │ │ │ ┌── set!                                                                       ;<--[B]
 12: │ │ │ │ │ └── #<procedure:set!>                                                          ;<--[B]
 12: │ │ │ │ │ ┌── k                                                                          ;<--[B]
 13: │ │ │ │ │ └── #<continuation:0>                                                         ;<--[B]
 13: │ │ │ │ │ ● the-continuation → #<continuation:0>    ; The bound value of the-continuation was    ;<--[B]
 13: │ │ │ │ └── #<unspecified>                           ; changed to the continuation (with i → 0).  ;<--[B]
 14: │ │ └── #<unspecified>
 14: │ │ ┌── (set!  i (+ i 1))
 14: │ │ │ ┌── set!
 15: │ │ │ └── #<procedure:set!>
 15: │ │ │ ┌── (+ i 1)
 15: │ │ │ │ ┌── +
 16: │ │ │ │ └── #<procedure:+>
 16: │ │ │ │ ┌── i
 17: │ │ │ │ └── 0
 17: │ │ │ │ ┌── 1
 18: │ │ │ │ └── 1
 19: │ │ │ └── 1
 19: │ │ │ ● i → 1            ; The binding was modified to i → 1.
 20: │ │ └── #<unspecified>    ; (It reflects to the environment of the continuation bound to the-continuation.)
 20: │ │ ┌── i
 21: │ │ └── 1
 21: │ └── 1
 22: └── 1
#<unspecified>
>
 22: ┌── (the-continuation)
 22: │ ┌── the-continuation
 23: │ └── #<continuation:0>
 24: ┌┬┬──── #<continuation:0>       ; The continuation was invoked! (ViiVi assumes #<unspecified>    <--[A]
 24: │ │ └── #<unspecified>           ; as the argument missed in invoking a continuation.)
 24: │ │ ┌── (set!  i (+ i 1))
 24: │ │ │ ┌── set!
 25: │ │ │ └── #<procedure:set!>
 25: │ │ │ ┌── (+ i 1)
 25: │ │ │ │ ┌── +
 26: │ │ │ │ └── #<procedure:+>
 26: │ │ │ │ ┌── i
 27: │ │ │ │ └── 1              ; The invoked continuation has an environment with the binding i → 1.
 27: │ │ │ │ ┌── 1
 28: │ │ │ │ └── 1
 29: │ │ │ └── 2
 29: │ │ │ ● i → 2            ; The binding was modified to i → 2.
 30: │ │ └── #<unspecified>    ; (It reflects to the environment of the continuation bound to the-continuation.)
 30: │ │ ┌── i
 31: │ │ └── 2
 31: │ └── 2
 32: └── 2
2
>
 32: ┌── (the-continuation)
 32: │ ┌── the-continuation
 33: │ └── #<continuation:0>
 34: ┌┬┬──── #<continuation:0>       ; The continuation was invoked!
```

```
  34: │ │ └── #<unspecified>
  34: │ │ ┌── (set!  i (+ i 1))
  34: │ │ │ ┌── set!
  35: │ │ │ └── #<procedure:set!>
  35: │ │ │ ┌── (+ i 1)
  35: │ │ │ │ ┌── +
  36: │ │ │ │ └── #<procedure:+>
  36: │ │ │ │ ┌── i
  37: │ │ │ │ └── 2              ; The invoked continuation has an environment with the binding i → 2.
  37: │ │ │ │ ┌── 1
  38: │ │ │ │ └── 1
  39: │ │ │ └── 3
  39: │ │ │ ● i → 3            ; The binding was modified to i → 3.
  40: │ │ └── #<unspecified>    ; (It reflects to the environment of the continuation bound to the-continuation.)
  40: │ │ ┌── i
  41: │ │ └── 3
  41: │ └── 3
  42: └── 3
3
>
  42: ┌── (define another-continuation the-continuation)
  42: │ ┌── define
  43: │ └── #<procedure:define>
  43: │ ┌── the-continuation
  44: │ └── #<continuation:0>
  44: │ ○ another-continuation → #<continuation:0>       ; The symbol another-continuation was
  45: └── #<unspecified>                                 ; initialized to the continuation being
#<unspecified>                                           ; bound to the symbol the-continuation.
>
  45: ┌── (test)                      ; To invoke test from the beginning.
  45: │ ┌── test
  46: │ └── #<closure:  []
  46: │       [(let ((i 0))
  46: │         (call/cc (lambda (k) (set!  the-continuation k)))>
  46: │         (set!  i (+ i 1))
  46: │         i)]>
  46: │ ┌── (let ((i 0))
  46: │ │     (call/cc (lambda (k) (set!  the-continuation k)))
  46: │ │     (set!  i (+ i 1))
  46: │ │     i)
  46: │ │ ┌── let
  47: │ │ └── #<procedure:let>
  47: │ │ ┌── 0
  48: │ │ └── 0
  48: │ │ ○ i → 0              ; The symbol i was initialized to 0 (since test was invoked from the beginning).
  48: │ │ ┌── (call/cc (lambda (k) (set!  the-continuation k)))
  48: │ │ │ ┌── call/cc
  49: │ │ │ └── #<procedure:call-with-current-continuation>
  49: │ │ │ ┌── (lambda (k) (set!  the-continuation k))
  49: │ │ │ │ ┌── lambda
  50: │ │ │ │ └── #<procedure:lambda>
  51: │ │ │ └── #<closure:  [(k)] [(set!  the-continuation k)]>
  51: │ │ │ ○ k → #<continuation:1>        ; The symbol k was init'ed to another cont. (with i → 0).     ;<--[B]
  51: │ │ │ ┌── (set!  the-continuation k)                                                               ;<--[B]
  51: │ │ │ │ ┌── set!                                                                                   ;<--[B]
  52: │ │ │ │ └── #<procedure:set!>                                                                      ;<--[B]
  52: │ │ │ │ ┌── k                                                                                      ;<--[B]
  53: │ │ │ │ └── #<continuation:1>                                                                      ;<--[B]
  53: │ │ │ │ ● the-continuation → #<continuation:1>    ; The bound value of the symbol                  ;<--[B]
  53: │ │ │ └── #<unspecified>                           ; the-continuation was changed                   ;<--[B]
  54: │ │ └── #<unspecified>                             ; to the continuation at the current point.      ;<--[B]
  54: │ │ ┌── (set!  i (+ i 1))
  54: │ │ │ ┌── set!
  55: │ │ │ └── #<procedure:set!>
  55: │ │ │ ┌── (+ i 1)
```

```
  55: │ │ │ │ ┌── +
  56: │ │ │ │ └── #<procedure:+>
  56: │ │ │ │ ┌── i
  57: │ │ │ │ └── 0
  57: │ │ │ │ ┌── 1
  58: │ │ │ │ └── 1
  59: │ │ │ └── 1
  59: │ │ │ ● i → 1                ; The binding was modified to i → 1.
  60: │ │ └── #<unspecified>       ; (It reflects to the environment of the continuation bound to the-continuation.)
  60: │ │ ┌── i
  61: │ │ └── 1
  61: │ └── 1
  62: └── 1
1

>
  62: ┌── (the-continuation)
  62: │ ┌── the-continuation
  63: │ └── #<continuation:1>
  64: ┌┬┬┬┬┬┬ #<continuation:1>   ; The continuation was invoked!                    <--[A]
  64: │ │ └── #<unspecified>
  64: │ │ ┌── (set!  i (+ i 1))
  64: │ │ │ ┌── set!
  65: │ │ │ └── #<procedure:set!>
  65: │ │ │ ┌── (+ i 1)
  65: │ │ │ │ ┌── +
  66: │ │ │ │ └── #<procedure:+>
  66: │ │ │ │ ┌── i
  67: │ │ │ │ └── 1
  67: │ │ │ │ ┌── 1
  68: │ │ │ │ └── 1
  69: │ │ │ └── 2
  69: │ │ │ ● i → 2                ; The binding was modified to i → 2.
  70: │ │ └── #<unspecified>       ; (It reflects to the environment of the continuation bound to the-continuation.)
  70: │ │ ┌── i
  71: │ │ └── 2
  71: │ └── 2
  72: └── 2
2
>
  72: ┌── (another-continuation)   ; To invoke a continuation bound in 42:∼44: to another-continuation
  72: │ ┌── another-continuation   ; (This continuation has an environment with a binding i → 3).
  73: │ └── #<continuation:0>
  74: ┌┬┬┬┬┬┬ #<continuation:0>   ; The other continuation was invoked!              <--[A]
  74: │ │ └── #<unspecified>
  74: │ │ ┌── (set!  i (+ i 1))
  74: │ │ │ ┌── set!
  75: │ │ │ └── #<procedure:set!>
  75: │ │ │ ┌── (+ i 1)
  75: │ │ │ │ ┌── +
  76: │ │ │ │ └── #<procedure:+>
  76: │ │ │ │ ┌── i
  77: │ │ │ │ └── 3                ; The invoked continuation has an environment with the binding i → 3.
  77: │ │ │ │ ┌── 1
  78: │ │ │ │ └── 1
  79: │ │ │ └── 4
  79: │ │ │ ● i → 4          ; The binding was modified to i → 4.
  80: │ │ └── #<unspecified>  ; (It reflects to the environment of the continuation bound to another-continuation.)
  80: │ │ ┌── i
  81: │ │ └── 4
  81: │ └── 4
  82: └── 4
4
>
/// Interpreter terminated normally.
$
```

Example 4

A simple sample code using `map` is input in the debugging mode.

```
> (map (lambda (x y) (/ x y)) '(1 2) '(3 4))
   0: ┌── (map (lambda (x y) (/ x y)) (quote (1 2)) (quote (3 4)))
   0: │ ┌── map
   1: │ └── #<procedure:map>
   1: │ ┌── (lambda (x y) (/ x y))
   1: │ │ ┌── lambda
   2: │ │ └── #<procedure:lambda>
   3: │ └── #<closure:  [(x y)] [(/ x y)]>
   3: │ ┌── (quote (1 2))
   3: │ │ ┌── quote
   4: │ │ └── #<procedure:quote>
   5: │ └── (1 2)
   5: │ ┌── (quote (3 4))
   5: │ │ ┌── quote
   6: │ │ └── #<procedure:quote>
   7: │ └── (3 4)
   7: │ │ ○ x → 1              ; x was initialized to 1.                    ; process (1)     ;<--[B]
   7: │ │ ○ y → 3              ; y was initialized to 3.                    ; process (1)     ;<--[B]
   7: │ │ ┌── (/ x y)                                                       ; process (1)     ;<--[B]
   7: │ │ │ ┌── /                                                           ; process (1)     ;<--[B]
   8: │ │ │ └── #<procedure:/>                                              ; process (1)     ;<--[B]
   8: │ │ │ ┌── x                                                           ; process (1)     ;<--[B]
   9: │ │ │ └── 1                                                           ; process (1)     ;<--[B]
   9: │ │ │ ┌── y                                                           ; process (1)     ;<--[B]
  10: │ │ │ └── 3                                                           ; process (1)     ;<--[B]
  10: │ │ └── 1/3                ; the result for (/ 1 3)                   ; process (1)     ;<--[B]
  10: │ │ ○ x → 2              ; x was initialized to 2.                    ; process (2)     ;<--[B]
  10: │ │ ○ y → 4              ; y was initialized to 4.                    ; process (2)     ;<--[B]
  10: │ │ ┌── (/ x y)                                                       ; process (2)     ;<--[B]
  10: │ │ │ ┌── /                                                           ; process (2)     ;<--[B]
  11: │ │ │ └── #<procedure:/>                                              ; process (2)     ;<--[B]
  11: │ │ │ ┌── x                                                           ; process (2)     ;<--[B]
  12: │ │ │ └── 2                                                           ; process (2)     ;<--[B]
  12: │ │ │ ┌── y                                                           ; process (2)     ;<--[B]
  13: │ │ │ └── 4                                                           ; process (2)     ;<--[B]
  13: │ │ └── 1/2                ; the result for (/ 2 4)                   ; process (2)     ;<--[B]
  14: └── (1/3 1/2)              ; map packed the two results into a list.
(1/3 1/2)
>
```

As shown above, there are two exceptions for the trace lines that are not closed, not connecting S-expressions and their evaluated values.

- A case that a continuation is invoked.
  It is occurring at the places with the comment `;<--[A]` in the Example 3 above. When a continuation is invoked, all the trace lines showing the evaluating processes so far are cut-off. The argument given to the continuation invocation is handed to the innermost trace line of the continuation, and the evaluation process restarts.

- A case that a higher-order procedure (e.g., `apply`, `call/cc`, `for-each`, or `map`) is invoked.
  It is occurring at the places with the comment `;<--[B]` in the Example 3 and Example 4 above. As these procedures invoke another procedure given as an argument, an extra trace line for the internal procedure invocation is shown. Since such a trace line neither has its input S-expression nor the evaluated value, no confusion should occur. If it is still confusing, the extra trace line would be changed to show the input S-expression and the evaluated value as well as the other trace lines in the future version.

Example 5

To see modification of a data structure, a simple code for generating a circular list is input in the debugging mode.

```
> (define list-A '(10 20 30))
    0: ┌── (define list-A (quote (10 20 30)))
    0: │ ┌── define
    1: │ └── #<procedure:define>
    1: │ ┌── (quote (10 20 30))
    1: │ │ ┌── quote
    2: │ │ └── #<procedure:quote>
    3: │ └── (10 20 30)
    3: │ ○ list-A → (10 20 30)            ; The symbol list-A was bound to a proper list (10 20 30).
    4: └── #<unspecified>                 ; (A new binding generation is shown with the marks ○ and →.)
#<unspecified>
   > (set-cdr!  (cddr list-A) list-A)
    4: ┌── (set-cdr!  (cddr list-A) list-A)
    4: │ ┌── set-cdr!
    5: │ └── #<procedure:set-cdr!>
    5: │ ┌── (cddr list-A)
    5: │ │ ┌── cddr
    6: │ │ └── #<procedure:cddr>
    6: │ │ ┌── list-A
    7: │ │ └── (10 20 30)
    8: │ └── (30)
    8: │ ┌── list-A
    9: │ └── (10 20 30)
    9: │ ■ (30) ⇒ #0=(30 10 20 .  #0#)    ; The list (30) was modified into a circular list #0=(30 10 20 .  #0).
   10: └── #<unspecified>    ; (Modification of the internal structure of a datum is shown with the marks ■ and ⇒.)
#<unspecified>
> list-A
   10: ┌── list-A
   11: └── #0=(10 20 30 .  #0)       ; The symbol list-A is now bound to a circular list #0=(10 20 30 .  #0).
#0=(10 20 30 .  #0)
>
```

11.2 Configuring evaluation environment with base level

"Base level" is a local term expediently introduced only for Viivi. It means an evaluation environment customized by the user arranging an ensemble of predefined object (procedure and constant) groups.

The purpose of this feature is to provide the user with an arbitrary subset of Viivi/R5RS evaluation environment and to let the user conduct various experiments in the environment. For example, trying to write codes in a still simpler evaluation environment (such that similar to Lisp 1.5 environment or an extremely simple one only with the procedures of `car`, `cdr`, and `cons`) must be an interesting puzzle, although it may not be practical. No languages other than Scheme/R5RS are suited for such an experiment[17].

The base level generated at boot time can be customized (but only this base level cannot be abandoned), and other customized base levels can be pushed/popped at any time during Viivi runs. By default, the base level generated at boot time includes all the predefined objects of Viivi.

Any procedure or constant predefined in Viivi belongs to one of the following groups:

`1`	A group of the objects defined in R5RS as	`syntax` (and so forth).
`2`		`library-syntax`
`4`		`procedure`
`8`		`library-procedure`
`16`		`optional-procedure`
`32`	A group of Viivi-specific objects	

User can combine these groups arbitrarily into a base level in Viivi. The bit summation of the integer value representing each group is used to specify the base level.

Example

To specify a base level composed of the predefined object groups defined as `syntax` and `procedure` in R5RS, use an integer value 5, that is the sum of integer `1` representing `syntax` and integer `4` representing `procedure`.

Example

To specify a base level composed of the predefined object groups other than those defined as `library-procedure` and `optional-procedure` in R5RS, use an integer value `N` integer 39, that is the bit-summation of those excluding integer values `8` and `16` representing `library-procedure` and `optional-procedure`, respectively.

Specification, generation, and deletion of a base level can be performed in the following ways:

- To specify a base level of `<N>` at boot time

[17]The user may find stupid the too much structure of a stack for the multiple base levels. Only a single set of the base level might be enough. This function is just a result of the stack structure convenient for implementing a flexible boot-up process of Viivi that is left as it is.

  - ◦ in the configuration files `(viivi-set! push-base-level <N>)`
  - ◦ at the head on the command line argument `-B <N>`

    - When there exist multiple configuration files, only the base level specified in the latest file is effective.
    - When there exist multiple descriptions in a configuration file, only the final description is effective.
    - When a base level specification is provided on the command line, the specified base level will be generated and `push`ed on top of the default base level unless it is specified at the first command line argument option.

- To generate a new base level of `<N>` during run time

  - ◦ using the procedure invocation `(viivi-set! push-base-level <N>)`
  - ◦ using the command line argument option `-B <N>`

- To abandon the current base level during run time

  - ◦ using the procedure invocation `(viivi-set! pop-base-level)`
  - ◦ using the command line argument option `-Bx`

    - Base levels generated during run time can be abandoned, although only the first base level generated at boot time cannot be abandoned.

On the current base level, a predefined object can be individually appended and removed. It is performed by invoking the procedure `viivi-set!`. Currently no corresponding command line argument options are prepared. The predefined object is specified with the symbol `<PREDEFSYM>` representing it. Don't forget to quote the symbol when it is directly specified, as the second argument for the procedure `#<procedure:viivi-set!>` is always evaluated.

- To append an individual predefined procedure on the current base level, use the S-expression
  `(viivi-set! append-predefined-symbol <PREDEFSYM>)`
- To remove an individual predefined procedure on the current base level, use the S-expression
  `(viivi-set! remove-predefined-symbol <PREDEFSYM>)`

Example

```
(viivi-set! append-predefined-symbol 'cos)
```

When the symbol `cos` did not have a binding on the current base level to the procedure object `#<procedure:cos>` (defined in R5RS to obtain the value of a mathematical function cosine), then the symbol is newly bound to the object `#<procedure:cos>` on the current base level.

Symbols with bindings on the currently base level can be shown by the procedure invocation `(viivi-property 'predefined-symbols)`.
The ever-built-in system procedures of Viivi, such as `viivi-title`, `viivi-set!` and `viivi-property`,

etc., are not shown.

Even when multiple base-levels are generated, each predefined object (a constant or a procedure) has always only one instance. Each predefined object on all the base-levels uses a reference to a common instance to suppress increase of occupied memory capacity.

11.3 Operations on circular lists

Display style for circular lists follows srfi-38. A warning message is displayed when a circular list is given as an argument to a procedure which expects the argument a proper list (in case of `verbose-level` is 3 or larger). Length of a circular list is defined as the negative of the number of the actual elements on the list.

Example

```
          > (define list-A '(10 20 30))
          #<unspecified>
          > (set-cdr!  (cddr list-A) list-A)
          #<unspecified>
          > list-A
A>>       #0=(10 20 30 .  #0#)
          > (length list-A)

B>>       /// >>Warning<<
B>>       /// #<procedure:length> found a circular list.
B>>       /// A negative value is returned..
B>>       /// Continuing the process...
C>>       -3
```

- `A>>`: Display style for circular lists follows srfi-38.
- `B>>`: Shows warning message when a circular list is given as an argument to a procedure (in case of `verbose-level` is 3 or larger).
- `C>>`: Length of a circular list is the negative of the number of the actual elements on the list.

11.4 Display styles for objects

Display styles for some objects are as follows:

- closure:

  `#<closure:` [argument part] [body ...]`>`

  Example

  The string expression for the closure defined by

  ```
  (lambda (n) (if (< n 1) 1 (* n (!  (- n 1)))))
  ```

  is

  ```
  #<closure:  [(n)] [(if (< n 1) 1 (* n (!  (- n 1))))]>.
  ```

- syntax:[18]

  `#<syntax:` [literal, ...] [{pattern}`-->`{template}, ...]`>`

  Example

  The string expression for the syntax **and** defined in R5RS "7.3. Derived expression types" is (after being formatted)

  ```
  #<syntax:  [] [{(_)}-->{#t},
                 {(_ test)}-->{test},
                 {(_ test1 test2 ...)}-->{(if test1 (and test2 ...)  #f)}]>.
  ```

- continuation:

  `#<continuation:<ID>>`

  An ID-number `<ID>` (starting from `0`) is shown after a colon character, which was sequentially assigned to each continuation at its generation time. It enables a specific continuation to be identified among multiple continuations.

  Example

  ```
  #<continuation:0> ;      the continuation firstly generated
  #<continuation:5> ;      the continuation generated as the sixth one
  ```

[18] Using curly braces not as "pattern`-->`template" but as "{pattern}`-->`{template}" is to indicate explicitly that "`-->`" is neither a literal nor a symbol included in the user's code.

11.5 Direct input for objects

Viivi can be specified to accept direct expressions for the Scheme objects independent of the symbol-binding relationship.

There are two ways to enable the direct input method.

- as the command line argument option:
  `-D <N>`
- as the procedure invocation:
  `(viivi-set! direct-input-objects <N>)`

`<N>` is a bit-summation of integer values `1,2,4`, each of which means

`1` to accept direct-input expressions for the predefined objects.
The predefined objects are

- predefined constants
  represented as `#<constant:`predefined-constant-name`>`

  Example `#<constant:PI>` Viivi-predefined constant for π

- predefined procedures
  represented as `#<constant:`predefined-procedure-name`>`

  Examples
  `#<procedure:let>` R5RS-predefined procedure `let`
  `#<procedure:cos>` R5RS-predefined procedure `cos` in R5RS
  `#<procedure:viivi-set!>` Viivi-predefined procedure `viivi-set!`

`2` to accept the direct-input expression for the unspecified value `#<unspecified>`

`4` to accept the direct-input expression for a nil `#<nil>`[19]

[19] This item may be deleted in the future versions, as it is not necessary if the usual S-expression `()` is used.

The purposes of introducing this option are as follows:

a. To specify predefined objects (predefined constants and procedures) directly independent of the symbol-binding relationship[20].

   There is a problem that, when a symbol-binding is modified at the top-level after defining a closure that uses the symbol, then the behavior of the closure becomes different from that was expected at the definition time.
   For example,

```
1> (define (func arg) (if arg 100 200))   ; definition for a closure
2> (func #t)             ; gives 100.  No problem.
3> (define if list)      ; redefining 'if' as 'list' at the top-level
4> (func #t)             ; gives (#t 100 200).  Should it give 100?
```

   Such a problem can be avoided by using the direct-input for the predefined procedure `if` at the time of the closure definition.

```
1> (define (func arg) (#<procedure:if> arg 100 200))
                         ; the predefined procedure 'if' is directly specified
2> (func #t)             ; gives 100.  No problem.
3> (define if list)      ; redefining 'if' as 'list' at the top-level
4> (func #t)             ; still gives 100 as expected.
```

b. To repair the symbol-binding modified at the top-level.

   When a binding of the symbol representing a predefined object is modified at the top-level, the symbol would be never correctly used. But the direct-input for the predefined object can be used to repair the symbol binding.

```
1> (define cos -)        ; When the symbol 'cos' was rebound to '-'
2> (cos 30)              ; gives -30.  'cos' was changed into '-'!
3> (define cos #<procedure:cos>)     ; repairing the symbol-binding
4> (cos 30)              ; gives 0.15425144988758405
                         ; 'cos' is now repaired
```

[20]For example, in implementing a closure, it could be possible to rewrite all the symbols used in the closure with their binding values at the definition time. However, I (personally) think that the referencing mechanism based on the symbol-binding even in the closure would make the interpreter more flexible, and hence this style was adopted by default in Viivi. The user can choose a robust usage with Viivi's closure rather than a flexible one, using the direct-input for the objects presented here.

11.6 Investigating macro pattern matching with the longest matching method

When multiple ellipses appear in series within a list or a vector in syntax definitions, Viivi investigates macro pattern matching with the longest matching method.

Example A

macro pattern of `macro-A`:	`(_ (e1 e2) ... e3 ...)`
user code:	`(macro-A (10 20) (30 40) 50 60 70)`

The macro pattern `(e1 e2) ...` matches `(10 20)` `(30 40)`, the longest part repeating the pattern `(e1 e2)` from the head of the argument `((10 20) (30 40) 50 60 70)` given for `macro-A` in the user code.
And the rest macro pattern `e3 ...` matches `50` `60` `70` in the rest of the user code. And therefore the pattern matching succeeds as the result.

Example B

macro pattern of `macro-B`:	`(_ (e1 e2) ... e3 ...)`
user code:	`(macro-B (10 20) 30 (40 50) 60)`

The macro pattern `(e1 e2) ...` matches `(10 20)`, the longest part repeating the pattern `(e1 e2)` from the head of the argument `((10 20) 30 (40 50) 60)` given for `macro-B` in the user code.
And the rest macro pattern `e3 ...` matches `30` `(40 50)` `60` in the rest of the user code. And therefore the pattern matching succeeds as the result.

Example C

macro pattern of `macro-C`:	`(_ (e1 e2) ... (e3 e4 e5) ... (e6 e7) ...)`
user code:	`(macro-C (10 20 30) (40 50 60))`

The first macro pattern `(e1 e2) ...` matches "zero elements", the longest part repeating the pattern `(e1 e2)` from the head of the argument `((10 20 30) (40 50 60))` given for `macro-C` in the user code.
The second macro pattern `(e3 e4 e5) ...` matches `(10 20 30) (40 50 60)`, the longest part repeating the pattern `(e3 e4 e5)` from the head of the rest argument `((10 20 30) (40 50 60))` given for `macro-C` in the user code.
And the third macro pattern `(e6 e7) ...` matches "zero elements", because no arguments are left in the user code. And therefore the pattern matching succeeds as the result.

Example D

macro pattern of `macro-D`:	`(_ (e1 e2) ... (e3 e4 e5) ... (e6 e7) ...)`
user code:	`(macro-D (10 20) (30 40) (50 60 70) 80 90)`

The first macro pattern `(e1 e2) ...` matches `(10 20) (30 40)`, the longest part repeating the pattern `(e1 e2)` from the head of the argument `((10 20) (30 40) (50 60 70) 80 90)` given for `macro-D` in the user code.
The second macro pattern `(e3 e4 e5) ...` matches `(50 60 70)`, the longest part repeating the pattern `(e3 e4 e5)` from the head of the rest argument `((50 60 70) 80 90)` given for `macro-D` in the user code.
And the third macro pattern `(e6 e7) ...` tries to match the rest argument `(80 90)` given for `macro-D` in the user code, but it fails. And therefore the pattern matching fails as the result.

Example E

macro pattern of `macro-E`:	`(_ e1 ... (e2 e3) ...)`
user code:	`(macro-E 10 20 30 (40 50) (60 70))`

The macro pattern `e1 ...` matches `(10 20 30 (40 50) (60 70))`, the longest part repeating the pattern `e1` from the head of the argument `(10 20 30 (40 50) (60 70))` given for `macro-E` in the user code.
The second macro pattern `(e2 e3) ...` matches "zero elements", because no arguments are left in the user code. And therefore the pattern matching succeeds as the result[21].

Example F

macro pattern of `macro-F`:	`(_ e1 ... (e2 e3) ...)`
user code:	`(macro-F 10 20 30 (40 50) (60 70 80))`

The macro pattern `e1 ...` matches `10 20 30 (40 50) (60 70 80)`, the longest part repeating the pattern `e1` from the head of the argument `(10 20 30 (40 50) (60 70 80))` given for `macro-F` in the user code. The second macro pattern `(e2 e3) ...` matches "zero elements". And therefore the pattern matching succeeds as the result[22].

[21] Introducing pattern matching based on the longest matching method, the matched pattern may be actually different from that expected from user's intuition such that the pattern `e1 ...` would match `10 20 30` and the pattern `(e2 e3) ...` would match `(40 50) (60 70)`.

[22] Introducing pattern matching based on the longest matching method, a macro pattern which seems not to match the user code at a glance (the pattern `e1 ...` matches `10 20 30`, but the pattern `(e2 e3) ...` does not match `(40 50) (60 70 80)`) does actually match the user code.

Example G

macro pattern of `macro-G`: `(_ (e1 e2 e3) ... e4 ... (e5 e6) ...)`
user code: `(macro-G (10 20) (30 40) 50 (60 70 80))`

The first macro pattern `(e1 e2 e3) ...` matches "zero elements", the longest part repeating the pattern `(e1 e2 e3)` from the head of the argument `((10 20) (30 40) 50 (60 70 80))` given for `macro-G` in the user code.
The second macro pattern `e4 ...` matches `(10 20)` `(30 40)` `50` `(60 70 80)` from the head of the rest argument `((10 20) (30 40) 50 (60 70 80))` given for `macro-G` in the user code.
The third macro pattern `(e5 e6) ...` matches "zero elements", because no arguments are left in the user code. And therefore the pattern matching succeeds as the result[23].

When an ellipsis of a scalar element appears in a list or a vector within a macro pattern as shown in the above examples F and G (such as `e1 ...` or `e4 ...` in the respective examples), such an ellipsis pattern of a scalar element will match all the rest elements given in the user code.
Therefore, to investigate exactly the internal structures of ellipsis appearing in series in a list or a vector of a macro pattern, ellipsis patterns with elements of complex internal structures need to be put ahead of the list/vector and a scalar element need be put at the end of the list/vector, as shown in the next example.

Example H

macro pattern of `macro-H`: `(_ (e1 e2 e3) ... (e4 e5) ... e6 ...)`
user code: `(macro-H (10 20 30) (40 50) (60 70) 80 90 100)`

The first macro pattern `(e1 e2 e3) ...` matches `(10 20 30)`, the longest part repeating the pattern `(e1 e2 e3)` from the head of the argument `((10 20 30) (40 50) (60 70) 80 90 100)` given for `macro-H` in the user code.
The second macro pattern `(e4 e5) ...` matches `(40 50)` `(60 70)` from the head of the rest argument.
The third macro pattern `e6 ...` matches `80` `90` `100` from the head of the rest argument.
This pattern matching agrees with our intuition.

Examples I

The following patterns can be also correctly investigated.

macro pattern of `macro-I1`: `(_ e1 ... e2 e3)`
macro pattern of `macro-I2`: `(_ e1 ... (e2 e3))`
macro pattern of `macro-I3`: `(_ (e1 e2) ... e3)`
macro pattern of `macro-I4`: `(_ (e1 e2) ... e3 ... e4)`
macro pattern of `macro-I5`: `(_ (e1 e2) ... (e3 e4))`

[23]Note that, introducing pattern matching based on the longest matching method, a user code which does not seem to match the macro pattern actually matches like Example G.

11.7 To use a transcript output as another input

In the transcript files of Viivi, every output line is commented with a semi-colon character ";" inserted at the line head. Thanks to this feature, user can use the transcript output as another input to reproduce the same evaluation processes, later on the same host (using a transcript output file) or simultaneously on another host (by communicating over the network).

If you are a Scheme beginner, for example, it is perhaps a good idea to write in a local configuration file `viivi.conf` as

```
(transcript-on "mySessionLog.txt").
```

When you encountered an error during inputting S-expressions from the keyboard to Viivi, you can investigate what was wrong by quitting Viivi immediately and then feeding the transcript file to Viivi with the debug option `-g` so as to see what happened during the interacting.

But the following two points must be remembered when you use the transcript output:

- If you use the current transcript filename (`mySessionLog.txt` in the above example) as the input source filename, then new transcript output is appended to the same file, making Viivi do endless evaluations. To avoid such a problem, the transcript output file must have been renamed when it is used as the input source file as follows:

```
$ mv mySessionLog.txt error.scm      # Rename the transcript file.
$ viivi -g error.scm                 # Read the renamed input source file.
```

- When you continue to use a transcript file, new contents of transcript output are always appended to the file. It may make the size of the transcript file huge. Rename or delete the transcript file regularly. (For example, it is a good idea to write a script renaming or deleting the transcript file just before `java` command invocation in the file `viivi` or `viivi.bat`.)

12 Other tips for using Viivi

12.1 To shut down Viivi

When you want Viivi to stop running (showing no prompt), or when you cannot escape from Viivi, you can shut down Viivi by killing JVM with `[Control]+[C]`[24] on most platforms.

12.2 To stop and restart screen scrolling

You may miss the output from Viivi due to screen scrolling, especially when you are using the debug trace output. In such a case, you can control the screen scrolling with the following key strokes:

- `[Control]+[S]` to stop the screen scrolling
- `[Control]+[Q]` to restart the screen scrolling

These key strokes can be used not only for Viivi but also for other applications running on Unix shell-terminals and Windows `cmd` generally.

12.3 Standard input/output files

The standard input source, the standard output destination, and the standard error output destination, prepared by default on many platforms are accessible as system files of the platforms. Some major examples are as follows:

- for Unix-like platforms

standard input source	`/dev/stdin`
standard output destination	`/dev/stdout`
standard error output destination	`/dev/stderr`

- for Windows platforms

standard input source	`con`
standard output destination	`con`
standard error output destination	`err`

Example

The debug trace output, which is written to the standard error output destination by default, can be redirected to the standard output destination as follows:

```
$ viivi -g -e /dev/stdout        for Unix-like platforms
> viivi -g -e con                for Windows platforms
```

[24] pushing `C`-key with `Control`-key of the keyboard

13 Reporting bugs of Viivi

Users are expected to report to the author bugs and problems they found. Thank you very much in advance. Corrections on the messages from Viivi and the descriptions on the manual in English or German by the native speakers are also welcome.

13.1 How to report

As the guestbook system is currently unavailable on the web page, e-mail is the only way for bug report.

- Send your bug report via e-mail to the e-mail address
  `<ilma@j00.itscom.net>`
  from your own e-mail address or through "Send an e-mail to ilma" on ilma's homepage
  `<http://home.j00.itscom.net/ilma/index.html>`.
  - Don't forget to write your e-mail address or a contact method. Without it, I cannot reply to you.
  - Writing "`viivi bug report`" at the title of the e-mail is helpful to distinguish the mails for bug-reports from those of other messages.
  - The Scheme source code that is attached to your bug report is used only for debugging Viivi implementation. But I may not be able to protect the source code from others. Therefore, I would like to ask you to send me not an important source code but an insignificant one generating an equivalent problem.

13.2 Things to be reported (examples)

- An exception of the JVM is directly shown as "`Exception occurs ...`". Such an Exception generated during Viivi running must have been caught and processed as an error by Viivi. But it was not handled correctly in this case. Especially, wrong Scheme codes or wrong operations may make Viivi down dumping exceptions. Please report the Exception message.
- A system error ID `F###` is shown. It means that an error occurred, which should not occur, and it was not handled correctly in this case. Please report the system error ID.
- An error message is shown for input which should be evaluated normally. It may be a bug.
- A result different from an expected one is returned. It may be a bug.
- Viivi shows strange behavior. It may be a bug.
- Something to improve Viivi is found. It may be a help for improvement.
- Something you want concerning Viivi is found. It may be a help for improvement.

14 Updating Viivi

14.1 Update of the system files

Basically speaking, you only need to unpack the `zip` file of the new Viivi distribution package (hereafter, we call the file simply package) over the existing files in the Viivi system directory, to continue using the new version of Viivi in the same environment of yours so far. How to unpack the package is the same as those described in "2 Downloading and installing Viivi".

1. If you have edited the boot-up script file (`viivi` or `viivi.bat`) before, and if you cannot find a back-up file (`viivi.SAVE` or `viivi.bat.SAVE`) of your current boot-up script file in the Viivi system directory, then you need to make a copy of your current boot-up script file to protect it from being overwritten by the files included in the package. To do so, do the following operation in the Viivi system directory before unpacking the new package:

   - for Unix-like platforms with `sh` family
     ```
     $ cp viivi viivi.SAVE
     ```
   - for Unix-like platforms with `csh` family
     ```
     % cp viivi viivi.SAVE
     ```
   - for Windows platforms
     ```
     > copy viivi.bat viivi.bat.SAVE
     ```

2. Unpack the new package into the Viivi system directory.

3. If you find a back-up file (`viivi.SAVE` or `viivi.bat.SAVE`) in the Viivi system directory, do the following operation in the Viivi system directory after unpacking the new package.

   - for Unix-like platforms with `sh` family
     ```
     $ cp viivi.SAVE viivi
     ```
   - for Unix-like platforms with `csh` family
     ```
     % cp viivi.SAVE viivi
     ```
   - for Windows platforms
     ```
     > copy viivi.bat.SAVE viivi.bat
     ```

14.2 Update of the copyright string

For the new versions of Viivi, it is enough with one copyright description

```
(viivi-set!  copyright "COPYRIGHT-STRING")
```

in the configuration file in the Viivi system directory

```
${VIIVI_HOME}viivi.conf.
```

Leave the copyright description as it is.

The user no more needs to put the copyright description in the user-editable configuration files

```
${HOME}/viivi.conf
${HOME}/.viivirc
${PWD}/viivi.conf
```

or

`${PWD}/.viivirc`.

Delete the old copyright descriptions in these configuration files, or they will cause a system error.

15 De-installing Viivi

To delete Viivi completely from the platform, do the following operations. Remark that Viivi no more runs, if you do the following operations.

Here, you are going to delete files and directories. Before doing the operations, make sure that those files and directories are no more necessary for you.

In the following, the home folder on Windows platforms is assumed to `C:\Users\ilma`. Adjust the underlined parts to your own home folder.

15.1 Deleting Viivi system files

First, delete all the system files of Viivi.

- Unix-like platforms with `sh`-family

```
$ cd ~/Viivi
$ rm 00readme*.txt Viivi.jar viivi*
$ cd ..
```

- Unix-like platforms with `csh`-family

```
$ cd ~/Viivi
$ rm 00readme*.txt Viivi.jar viivi*
$ cd ..
```

- Windows platform

```
> C:
> cd \Users\ilma\Viivi
> del 00readme*.txt Viivi.jar viivi*
> cd ..
```

15.2 Deleting Viivi system directory

And then, delete Viivi system directory ${`VIIVI_HOME`} where the Viivi system files resided.

- Unix-like platforms with `sh`-family `$ rmdir ~/Viivi`
- Unix-like platforms with `csh`-family `% rmdir ~/Viivi`
- Windows platform `> rd C:\Users\ilma\Viivi`

If you cannot delete Viivi system directory with the above operation, there may exist files in it. After confirming that the files in the Viivi system directory are not important for you, delete them, and then try to delete Viivi system directory, again.

15.3 Delete user's work directory

Do operations similar to those performed on Viivi system directory also on user's work directory.

The user's work directory may contain important files that the user have created. If you have important files in it, make backups for the important files before deletion.

- Unix-like platforms with `sh`-family

```
$ cd ~/ViiviWork
$ rm UNNECESSARY_FILES
$ cd ..
$ rmdir ~/ViiviWork
```

- Unix-like platforms with `csh`-family

```
% cd ~/ViiviWork
% rm UNNECESSARY_FILES
% cd ..
% rmdir ~/ViiviWork
```

- Windows platforms

```
> C:
> cd \Users\ilma\ViiviWork
> del UNNECESSARY_FILES
> cd ..
> rd \Users\ilma\ViiviWork
```

15.4 Retracting Viivi environment variable settings

Finally, retract all the environmental variable settings for Viivi.

- Unix-like platforms with `sh`-family

  Delete the following four lines at the end of the environment configuration file (`.profile`, `.bashrc`, or `.kshrc`) that were appended in "4.2 Setting environmental variables":

```
VIIVI_HOME=~/Viivi                    #Remove this line!
export VIIVI_HOME                     #Remove this line!
PATH=${PATH}:${VIIVI_HOME}            #Remove this line!
export PATH                           #Remove this line!
```

  Then retract the environment variable settings for Viivi from the current environment:

```
$ VIIVI_HOME=
```

```
$ export VIIVI_HOME
$ .  ~/.profile          or      $ .  ~/.bashrc
                         or      $ .  ~/.kshrc
```

- Unix-like platforms with `csh`-family

  Delete the following two lines at the end of the environment configuration file (`.login` or `.cshrc`) that were appended in "4.2 Setting environmental variables":

```
set VIIVI_HOME ~/Viivi                    #Remove this line!
set PATH ${PATH}:${VIIVI_HOME}            #Remove this line!
```

  Then retract the environment variable settings for Viivi from the current environment:

```
% unset VIIVI_HOME
% source ~/.login       or      $ source ~/.cshrc
```

- Windows platforms

  For Windows platforms, the standard GUI method is shown here. If you know how to erase the environmental variables in the Windows platforms by editing system files directly, you may do it at your own risk.

```
[Start]-->[Control Panel]-->([System and Maintenance])-->[System]-->
[Tasks]-->[Advanced System Settings]-->[Environment Variables]
```

  Now you will see

Variable	Value
`VIIVI_HOME`	the-string-you-set-in-"4.2 Setting environmental variables"
`PATH`	existing-path-strings-here`;%VIIVI_HOME%`

  Select each with a mouse-click, then remove the setting choosing `-->[Delete]`.

  If you have set other strings to the value of the `PATH` variable for other applications, choose `Edit` instead and erase only the part "`;%VIIVI_HOME%`".

  Then select `-->[OK]` and reboot Windows to activate the new settings.

— End of the user's manual —